ON PROVING TERMINATION OF PROGRAMS

A *Thesis submitted in*
partial fulfilment of the requirements

for the
DEGREE OF

MASTER OF TECHNOLOGY

by
S. Ravindran

to the
Department of Electrical Engineering
INDIAN INSTITUTE OF TECHNOLOGY
KANPUR

A C K N O W L E D G E M E N T S

I am deeply indebted to Dr. S.K. [illegible] for his guidance and counsel during the course of this work. He found time and patience for many valuable discussions and has been a constant source of encouragement and inspiration. I am also extremely grateful to Prof. V.Rajaraman for many useful discussions. But for his help, encouragement and kindness, in so many ways, this work would not have been possible.

I wish to express my appreciation to Dr. C.R. Muthukrishnan for many interesting discussions; to Dr. H.N. Mahabala for his interest and encouragement and to Dr. V.Gupta for giving me the reprints of papers, he has collected, for long periods.

S. Ravindran.

ABSTRACT

A model of computation in a language like FORTRAN has been defined. It is well known that the problem of proving termination of programs, in its complete generality is recursively unsolvable. We have, hence, developed a heuristic algorithm for generating a sufficient condition for proving termination of large subclasses of programs within the framework of the given model of computation. The use of this condition and the techniques for proving the condition are illustrated.

CONTENTS

C H A P T E R 1

§ 1.1 Introduction

The central problems in the area of computer program analysis are as follows

1. Does the program solve the given problem?
2. Is the program equivalent to another? i.e. for all relevant data do both programs compute the same results?
3. Under what condition does the program not terminate? or does it terminate for all initial data?
4. Can we give a measure of efficiency for a program?
5. How large is the program? How much is the time estimation for a complete execution for one set of data?

These questions are extremely difficult to answer. They are closely connected with the theory of algorithims and the study of both general and specific properties of algorithims which are to be realised with a computer. Some, of the above questionslike (1), (2) and (3) are known to be unsolvable in their complete generality.

§ 1.2 Definitions & some known results

In this section, for the sake of continuity, we briefly review some of the results obtained in theory of computability and mathematical logic. We

also note some standard definitions in graph theory for later use.

The following definitions in the theory of computability have been taken from some standard texts (44), (45), and (46). We assume that the reader is familiar with the formal definition of an algorithm or effective procedure. A decision problem enquires as to the truth or falsity of each of a whole class of statements, and such a problem is (recursively) unsolvable if there is no algorithm which supplies all the answers. We say a decision problem is partially solvable if there is an effective enumeration of the true statements in the class, or equivalently, if there is an algorithm which when presented with a statement of the class which is true, comes to the correct decision, but otherwise fails to terminate.

Let T be any turing machine (TM) (44). We will define f_T, the function computed by T as follows. For any integer x, if the TM T when started on input tape containing x halts, and gives y as the output then $f_T(x) = y$. If T on input x does not halt $f_T(x)$ is undefined. If T halts on all inputs, then f_T is a total computable function. The class of all functions which

are Turing computable is called the class of partial recursive functions. If a partial recursive function is total it is called a recursive function. The characteristic function of the set A is a function f such that $f_A(x) = 0$ if $x \in A$

$= 1$ if $x \notin A$

A set is said to be recursive if its characteristic function is recursive. A property P on integers is said to be decidable if $A = \{ x \mid P(x) \}$ is recursive and undecidable (recursively unsolvable) otherwise.

Let T_x be the xth TM in the standard indexing of TM's (45). Let $\phi_x^{(k)}$ be the partial recursive function of k variables, computed by the T_x with index x. Following results are well known in the literature (44).

Theorem 1.2.1

HALTING PROBLEM:- It is recursively unsolvable to decide whether an arbitrary TM T when started on input x halts.

COROLLARY 1.2.3:- It is recursively unsolvable to decide whether an arbitrary TM T when started on blank tape halts.

Theorem 1.2.3:- It is recursively unsolvable to decide whether an arbitrary given, Turing machine T halts on all inputs.

Theorem 1.3:- It is recursively unsolvable to decide if for any x & y $\phi_x = \phi_y$.

Mathematical logic is quite extensively used in formalising and analysing the class of problems stated earlier. We now present some of the standard definitions needed in this context. These definitions have been taken from a standard text on Mathematical logic (47).

We assume the reader is familiar with the existential quantifier ($\exists$) and universal quantifier ($\forall$) and their usage. We shall use the symbols $\sim$ (negation) and $\supset$ (implication), commas, parenthesis, individual variables x_1, x_2...; individual constants a_1, a_2...; predicate letters A_1^1, A_1^2...; and function letters f_1^1, f_1^2... The superscript indicates the number of arguments.

The terms are defined as

(a) Variables and individual constants are terms

(b) If f_i^n is a function letter, and t_1, t_2 ... t_n are terms then f_i^n (t_1... t_n) is a term.

(c) An expression is a term only if it can be shown to be a term on the basis of clause (a) and (b).

The predicate letters applied to terms yield the atomic formulaei.e. if A_i^n is a predicate letter and t_1... t_n are terms, then A_i^n (t_1,...t_n) is an atomic formula.

The well formed formulas (wff) are defined as follows:

(a) Every atomic formula is a wff.

(b) If A & B are wff's then $(\sim A)$ or $(A \supset B)$ and $(\forall y\ A)$ are wffs

(c) An expression is a wff only if it can be shown to be a wff on the basis of clauses (a) & (b). In $(\forall y\ A)$, "A" is called the scope of the quantifier $\forall y$.

An occurance of a variable x is bound in a wff, iff, either it is the variable of a quantifier '$(\forall x)$' in the wff, or it is within the scope of a quantifier "$\forall x$" in the wff. Otherwise, the occurence is said to be free in the wff. If A is a wff and t is a term, then t is said to be free for x_i in A iff no free occurences of x_i in A lies within the scope of any quantifier $(\forall x_j)$ where x_j is a variable in t.

Wffs have meaning only when an interpretation is given to the symbols. An interpretation consists of a non-empty set D, called the domain of interpretation, and an assignment to each predicate letter A_j^n of an n-place relation in D, to each function letter f_j^n of an n place operation in D (i.e. a function from D^n into D), and to individual constant a_i some fixed element of D. Given

such an interpretation, the variables are thought of as ranging over D, and other symbols like $\sim, \vee, \wedge, \forall, \& \exists$ have their usual meaning. For a given interpretation, a wff without free variables (called a closed wff) stands for a relation on the domain of the interpretation which may be satisfied (true) for some values in the domain of free variables and not satisfied for the others. A formula P is valid iff P is true in every interpretation.

We assume the reader is familiar with the formal definitions of Axioms, Axiom system, Rules of inference, completeness, theorem and proof in a logical system. It is a well known result that the first order predicate calculus is complete and the problem of deciding whether any given formula is valid is undecidable.

Graph theory has been used quite extensively to describe the flow of control through a program. We now define the necessary graph theoretic terms for later use. A directed labelled graph G is a 5 - tuple (X, V, B, t, Ψ) where

X is a finite set of vertices

$V \subseteq X \times X$ is the set of (directed) arcs.

B is a finite set of (vertex) labels

$t : X \to B$ is the vertex labelling function.

$\Psi : V \to \{0, 1\}$ is the arc labelling function

Given a directed labelled graph G we can definte a multi-valued function $\Gamma : X \to X$ such that $(x, y) \in V \Rightarrow y \in \Gamma_x$

Henceforth, a graph denotes a directed labelled graph.

Let (a, b) be an arc 'a' is called the initial vertex & 'b' is called the terminal vertex of the arc (a, b)

C' is said to be a subgraph of $G = (X, V, B, t, \psi)$ if $X' \subseteq X$, V' is the restriction of V to $X' \times X'$, t' & ψ' are restrictions of t & ψ to X' & V' respectively.

A path μ is a sequence of vertices $x_1, x_2 \ldots x_n$ such that $(x_i, x_{i+1}) \in V$ for $1 \leq i \leq n-1$ The path μ is said to meet each of the vertices $x_1\ x_2 \ldots x_n$. We use the notation $x_j \in \mu$ to denote the fact that u meets x_j.

In the above definition of μ, x, is called initial vertex and x_n the final vertex of μ. A path is elementary if it does not meet the same vertex twice.

A circuit is a finite path $\mu = (x_1, x_2 \ldots x_n)$ in which the initial vertex x_1 coincides with x_n. A circuit is elementary if apart from the coincident initial and terminal vertex, every vertex which it meets is distinct.

The length of a path $\mu = (x_1, x_2 \ldots x_n)$ is the number of arcs in the path μ ($|\mu| = n-1$). For two vertices x & y of the graph we write $x < y$ if there is a path from x to y.

§ 1.3 MOTIVATION & AIM

One of the most important and time consuming phases of computer programming is to ensure that it works pro perly. Most of the present day processors go little farther than pointing out syntactic errors The way a programmer tackles this problem is by experienced intuition, for simple programs, supplemented by trial and often error for more complicated ones. It has been found that the time spent on program testing is more than half the entire time spent on the programming project. It is mot sufficient for a programmer to know that the program works most of the time or even that, so far, it has never made a mistake. The cost of removing errors after a program has gone into use is far greater, particularly in the case of items of computer manufacturer's software, for which large part of the expense is borne by the user. And finally, the cost of error in certain types of programs may be almostincalculable - a lost spacecraft, a collapsed building, a crashed aeroplane or a world war. To avoid these impending disasters, effective program checkout is imperative for any complex computer program. The real question, therefore, is whether a given program can be counted upon to fulfil its functional specifications successfully every single

time. The ideal thing, therefore, that one can hope for is that the verification of the correctness of a program be carried out by the computer itself. So that, when the correctness of the program, its compiler, and the hardware of the computer have all been established with a degree of mathematical certainty, it will be possible to place great reliance on the results of the program, and predict their properties with confidence limited only by the reliability of the electronics. As McCarthy (1) notes, that, inspite of the theoretical limitations discussed earlier ".....the limitations on what we have been able to make computers do so far clearly come far more from our weakness as programmers than from intrinsic limitations of the machine."

Our interest in this thesis is on a problem related to the correctness of a program, namely the problem of termination. That is, we are interested in finding out whether a given program terminates for all values of input data. Engler (18) and Manna (23) have shown that the successful verification of a property of the program can be normalised to the notion of termination of a program. It is well known that problem of deciding termination in its complete generality is recursively unsolvable. It can also be shown that there does not even exist a partial algorithem (44) for deciding termination of a program and hence what we can at most

to develop is an algorithem to tackle large interesting subclasses of program schemas. Our aim in this thesis is to develop a sufficient condition for termination. A substantial effort has recently been put into finding methods for proving termination, validity and equivalence. Hence for the sake of completeness we briefly survey, in the next section, the work reported in these areas.

§ 1.4 S U R V E Y

We have tried to classify the work based on the approach and the problem tackled.

Ianov's (11)paper is perhaps the earliest work reported on the theory of equivalence of algorithems. He considers 'logical schemes' representing the sequential and control properties of programs which remain when almost all the information about the nature of the basic operations are disregarded. He is able to obtain a complete decision procedure for the equivalence of schemes, but sacrifices a great deal of the essential structure of programs, leaving in effect, little more than a finite automata.

Kaplan (21) has developed a series of algorithms which always produce an answer YES or MAYBE when questioned as to the equivalence of two programs. The condition he has developed is that the sequence of operators in the regular expression constructed from program schema should be finite

state equivalent. This property is known to be decidable He claims that the class of schemata he has considered includes the logical schemata of Ianov in the sense that the arbitrary restriction placed by Ianov that no operator may appear twice in a logical schema has been removed.

Ershov (15) was the first to attempt a formalism for describing computation. Best part of his paper is directed towards establishing relationships between his "operator algorithms" and other formalisms, so that he gives little information regarding the equivalence theory of algorithms.

Mc Carthy (1) says that the theory of finite automata does not seem to be an adequate basis for the mathematical theory of computation because "the results which use finiteness of number of states tend not to be very useful in dealing with present computers, which have so many states that it is impossible for them to go through a substantial fraction of them in a reasonable time." He has pointed out that the results in the theory of computability are mainly negative in character. He advocates a study of mathematical theory of computation one of the main gains of which will be the elimination of debugging. His aim is also to give a theory for equivalence of computation processes. He introduces a formalism using conditional expressions, in which new functions are

produced from old functions by a recursive definition process. A computer program in his formalism, given in the form of a flow diagram, in which the flow of computation is controlled by conditional expressions, can be converted to a series of recursive function definitions. Various techniques are presented whereby such recursive functions, and hence programs may be proved equivalent.

By postulating (3) that "the meaning of a program is defined by its effect on the statevector" he has been able to show the correctness of a compiler for arithmetic expressions in a very elementary language.

Cooper (9) has generalised certain of McCarthy's equivalence proofs involving undefined functions, satisfying certain conditions and relations . These techniques seem to be applicable only to program, of a relatively simple nature.

Igarashi (12) has classified the statements in ALGOL into different types and for each type he has presented a set of axioms and the rules of inference which have been shown to be complete. The proofs for equivalence of even simple programs is long and cumbersome and also, as with all axiomatic approaches, there is not even a clue as to how the proof should proceed.

Hoare (14) has pointed out that the axioms for computer arithmetic is not the same as the arithmetic familiar to mathematicians. The axiom set and rules of inference chosen should be in the domain of the computer in that their use is confined to a finite set of integers and we need special axions to deal with special conditions like overflow etc. This paper is mainly in the nature of suggestions and does not contain any proven results.

Paterson (16) has given a procedure for deciding equivalence of loop free schematas, and schematas which always converge. He also considers schematas in which every statement is in no more than one loop, that is, no loops are nested, or interesecting, subject to the restriction that all functions are monadic. He has shown that the problem of equivalence is decidable for these schemata. He has also established that "... it is not necessary for the unsolvability for the schemata to have arbitrarily complicated flow structures, but it is an open problem to find how simple a structure will suffice."

Floyd (22) and Manna (23) have developed wellformed formulaes associated with program schemas, in some system of logic, such that the program schemas possesses a property if and only if the wellformed formula is shown to be satisfiable or valid in the system.

Floyd points out that the semantic definition of each operation in a programming language is most properly given as a logical rule which tells exactly what assertions can be proved after the operation, from what assertions are true beforehand. He labels each connection in the flowchart of a program about the current state at the time computation traverses that connection. The general method consists of proving for each box in the flowchart, that if for any one of the assertions on the arrows leading into the box is true before the operation in that box is performed, then all of the assertions on the arrows leaving away from the box are true after the operation. Once this has been proved for every box in the flowchart, then by induction on the number of boxes executed, one sees that if a program is entered by a connection whose associated assertion is true, it will be left, if at all, by a connection whose associated assertion is also true. He suggests that termination of a program can be proved by associating with each connection in the flow-chart a weight function taking its values in a well ordered set, such that after every execution of abox the weight function decreases. Floyd's technique of proving termination is necessarily closely linked with his technique for establishing the correctness of a program. Zohar Manna's (23) technique is a direct consequence of Floyd's proposition. That is, he associates

with every vertex in the abstract program a_i logical formula (semantic condition) and by taking the conjunction of all these formulas he obtains a well-formed formula in a system of logic which is unsatisfiable if and only if the program schema terminates. The formulae developed are long even for simple programs. It is well known that the satisfiability or validity of a wff is undecidable. Manna has not isolated subclasses of program schemas for which his technique will come out successfully.

Though Engler (18) has proved that the successful verification of the property of a program can be normalized to the termination of a program he has not dealt with techniques of establishing the same.

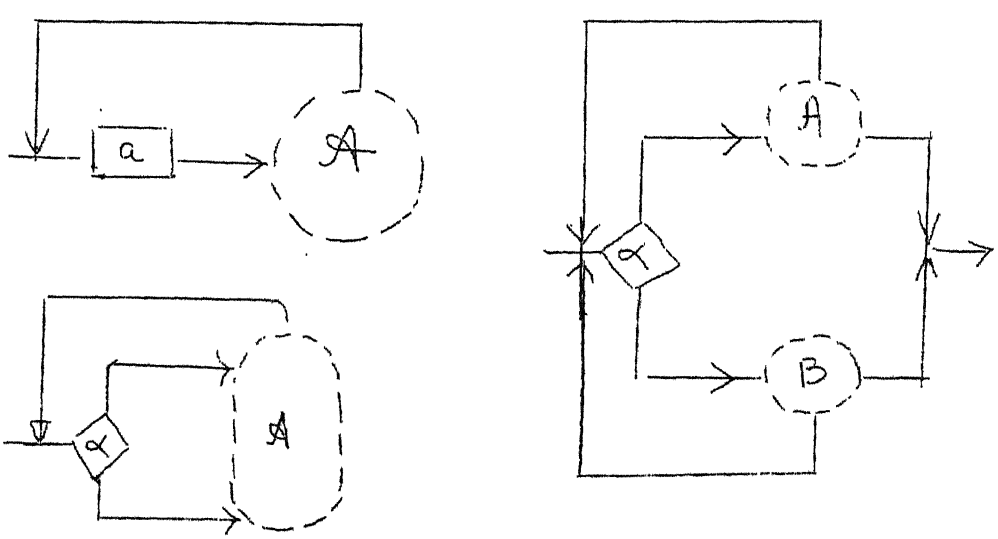
Another interesting approach to study the properties of program has been through the use of graph theory. The early attempts in this direction are due to R.T.Prosser (32), R.B.Marimont (33),S.Warshall(34), R.M. Karp (42), C.V.Ramamoorthy (50) and C.J .Meloney (36). They all took the connectivity matrix of the graph for analysis without looking at the semantics of statements. Consequently the results they have obtained are, as in graph theory, only regarding the structural flaws. Lee Krider (40), E.A. Voorhees (38)& M.Meakawa (43) have tried to construct a linear string from a

program with the aim of obtaining the flowchart of the program. Again the results obtained are not of any fundamental importance.

Cooper (8) has tried to give some set of graph transformation so that any given graph can be put in a standard form in which it is easy to prove results about the program by taking advantage of the specia features of the graph. An interesting question therefore is whether a particular transformation increases or decreases the complexity of the program in some sense. Basu (28) has shown that if the transformations suggested by Cooper are applied to nodes satisfying a certain safety criteria then the complexity of the cycle structure of the digraph measured in terms of its cycle rank (minimum no.of nodes to be deleted to make the graph cycle free) is held constant. He has also shown that these transfermations do not reduce the cycle rank. An interesting result proved by Paterson that must be noted here is that any attempt to make a rule book of transformations for transforming any two equivalent program schemas into one another must fail.

Bohm and Jacopini (41) have shown that it is not possible to decompose every given graph into a finite number of given base diagrams, but this becomes possible at the semantical level. They have shown that

for any flow diagram there is an equivalent one which is decomposable into two types of basic blocks, that is, a form in which all loops are properly nested. But, as pointed out by Cooper (10), that "if however one's motivation is to simplify the program structure so that we may better answer questions such as whether the program loops undefinitely then this transformation at the semantical level is of no help at all." We note here another result from Bohm and Jacopini's paper for later reference. They have shown that any program falls into one of the three following types:



-oOo-

CHAPTER 2

2.1 Introduction

In this chapter we present a model of computation which has been used in a later analysis. We formalise our intuitive notions about computation and other properties of programs. We make use of McCarthy's (1) metaformalism for defining various functions. Implicit use is also made of his axioms for manipulating the conditional expressions appearing in these definitions.

The computation model defined in this chapter intuitively depicts the source language computation. This not only makes the results machine independent but also allows direct interpretation of the results into real life programming languages. Similar models have been studied by Paterson (16), Kaplan (21) and Manna (23). Several simplifying assumptions have been made in defining the model. Such features of the source language as the use of subroutines, subscripted variables etc. have not been considered. However, input/output statements and conditional assignments have been included which are similar to those found in a language like FORTRAN. We next proceed to describe our idealized computation language, called an abstract computation language (ACL).

§ 2.2 ABSTRACT COMPUTATION LANGUAGE

The Roman numbers (0,1,...) and upper case letters (A,B,C,...) are used to denote individual constants (C). Variables (V) are denoted by lower case letters (a,b,..., x, u_1, u_2... v_1, v_2 ...); $p_1^{(k)}$, $p_2^{(k)}$.. $p_n^{(k)}$ stand for k - place predicate symbols; f, g, h..... are used to denote function symbols; logical connectives $\sim$ (negation) and v (disjunction) and $\wedge$ (conjunction) are also used.

2.2.1 FUNCTION TERM

(1) Any of the individual constants, variables or empty symbol ($\in$) is a function term.

(2) If $f(*_1, *_2, ...*)$ is an n-ary function symbol and t_1, t_2 ... t_n are function terms then $f(t_1, t_2...t_n)$ is also a function term;

(3) The only function terms are those that can be defined by (1) and (2) above.

2.2.2 SIMPLE PREDICATE TERM

If $p_i^{(k)}$ is a k-ary predicate symbol and x_1, $x_2..x_k$ are variable or constants then $p_i^{(k)}(x_1, x_2...x_k)$ is a Simple predicate term.

2.2.3 PREDICATE TERM

(1) Any simple predicate term is a predicate term.

(2) If $B(*_1, *_2...*_n)$ is an n-ary Boolean function

and $t_1, t_2 \ldots t_n$ are predicate terms then so is $B(t_1, t_2 \ldots t_n)$

(3) The only predicate terms are those that can be defined by (1) and (2) above.

We note that any predicate term can be expressed in the disjunctive normal form (d.n.f) in which each disjunct is a k-ary simple predicate term.

The equality sign is used in different contexts but their usage does not cause any confusion. Usual set theoretic notations are used.

2.2.4 A statement S in ACL is one of the following types.

(1) ASSIGNMENT STATEMENT (type A). This is of the form $u=\sigma$ where u is called the assigned variable and σ is a function term. u is assigned the value of σ. u is said to be dependent on the variables in σ.

(2) UNCONDITIONAL TRANSFER STATEMENT (Type UT). This is of the form $\uparrow_i$ where i is a positive integer. Control is transferred unconditionally to the i^{th} statement in the given sequence of statements.

(3) TEST STATEMENT (Type P). This is of the form $IF(P)\uparrow_i$. If the predicate term P is true then the i^{th} statement in the given sequence of statements is executed next, else the next sequential instruction is taken.

(4) CONDITIONAL ASSIGNMENT (Type CA). This is of the form $IF(P)u=\sigma$. If the predicate term P is true then the

assignment $u=\sigma$ is executed and then the next sequential instruction is taken, else the next sequential instruction is taken.

(5) INPUT (Type IP). This is of the form READ v_i. The effect of executing this statement is equivalent to assigning a constant value to the variable v_i.

(6) OUTPUT (Type OP) This is of the form PRINT v_i. It just prints out the value of the variable v_i.

(7) START (Type $\bar{I}$). It is a null statement.

(8) HALT STATEMENT (Type F). This is of the form STOP. This statement stops the computation.

2.2.5 A variable is an input variable if it occurs in a statement of the type IP and output variable if it occurs in a statement of the type OP.

2.3 PROGRAM SCHEMA AND FLOW GRAPH

2.3.1 PROGRAM SCHEMA (PS) $\Delta(\alpha, \beta)$ in the ACL is a finite ordered sequence of statements in which $\alpha \& \beta$ are the set of input and output variables respectively.

Given a PS $\Delta(\alpha, \beta)$ in ACL, we would like to explicitly denote the flow of control in the schema during computation. To this end, we define a flowgraph G_Δ for a PS $\Delta(\alpha, \beta)$ as follows.

2.3.2 The flowgraph G_Δ of a PS $\Delta(\alpha, \beta)$ is a labelled graph (X, V, B, t, Ψ) where

$B = (A, P, CA, UT, IP, OP, F, \bar{I})$

X : - finite set of vertices corresponding to the s statements in PS $\Delta(\alpha, \beta)$

$t: X \to B$ is a mapping which denotes the type of statement associated with the vertex x.

$\Psi : X \times X \to (0,1)$ is a labelling function assigning Boolean labels to some arcs.

then $V = \{(x,y) \mid x \in X, y \in \Gamma x\}$. is the set of arcs in G_Δ satisfying following conditions:

(a) If $t(x_i)$ = A or IP or OP

then $\Gamma x_i = x_{i+1}$; $\Psi(x_i, x_{i+1}) = \phi$

(b) If $t(x_i)$ = P

then $\Gamma x_i = (x_{i+1}, x_j)$

$\Psi(x_i, x_{i+1}) = 0$ & $\Psi(x_i, x_j) = 1$

(c) If $t(x_i)$ = CA

then $\Gamma x = (x_{i+1}, x_j)$; $\Psi(x_i, x_{i+1}) = 0$

$\Psi(x_i, x_j) = 1$; $\Gamma x_j = x_{i+1}$; $t(x_j) = A$;

$\Psi(x_j, x_{i+1}) = \phi$

(d) If $t(x_i)$ = UT

then $\Gamma x_i = x_j$; $\Psi(x_i, x_j) = \phi$

(e) If $t(x_i)$ = F then $\Gamma x_i = \phi$

(f) If $t(x_i) = \bar{I}$

then $\Gamma x_i = x_{i+1}$; $\Psi(x_i, x_{i+1})$ ϕ

2.3.3 START AND FINAL VERTICES

A null statement (type $\bar{I}$) is appended to the first statement in the PS Δ (α, β) in ACL. The vertex corresponding to this statement in G_Δ is called the start vertex x_I $\therefore \forall x \in X$, $(x, x_I) \notin V$. Any vertex x such that $t(x) = F$ is called a final vertex. We note that $\forall x \in X$, s.t. $t(x) = F, (x,y) \notin V$

From now on we consider only those PS's for which the flowgraph satisfies the following conditions

(1) $\forall x \in X, \ 1 \leq |\{\Gamma_x\}| \leq 2$

(2) $\forall x \in X, \ x_I < x$ & $x < x_F$) Continuity conditions

Intuitively, the continuity condition implies that for any vertex $x \in X$, there is a path from the start vertex to a final vertex meeting x.

2.3.4 Lemma: Let G_Δ be a flowgraph and let $\mu = x_{i_1}; x_{i_2}, x_{i_3}, \ldots x_{i_k}, x_{i_1}$ be an elementary circuit (?. 2). Then, there exists a pair of vertices y_1 and y_2 not contained in μ such that for some x_{i_m} and x_{i_n} , $1 \leq m \leq k$, $1 \leq n \leq k$ $(y_1, x_{i_m}) \in V$ and $(x_{i_n}, y_2) \in V$. Proof for this lemma is immediate from the continuity condition.

2.3.5 LOOP Let $S_1, S_2, S_3 \ldots S_k$ be an ordered sequence of statements in a PS $\Delta(\alpha, \beta)$ & let the corresponding

sequence of vertices in the flowgraph G_Δ be x_{i_1} , x_{i_2} .... x_{i_k} . Then, this ordered sequence of statements is said to constitute a (conditional) loop if and only if

(1) $x_{i_1} \in \Gamma x_{i_k}$.

(2) $t(x_{i_k}) = P$ or UT

(3) x_{i_1} , x_{i_2} ... x_{i_k} is an elementary path in G_Δ

Henceforth, a loop L, defined as above, will be described as $L = x_{i_1}$, x_{i_2} ... x_{i_k} .. x_i Note that this sequence of vertices is an elementary circuit.

2.3.6 In the above definition of a loop L, x_{i_1} is called the first vertex and x_{i_k} the last vertex of the loop L

2.3.7 A loopset $\bar{L}$ is the set of all loops with the same first and last vertices.

$$\bar{L} = \{ L^{(1)}, L^{(2)} \ldots L^{(j)} \ldots L^{(n)} \}$$

2.3.8. A vertex $x \in L^{(j)}$ is an EXP (of the loopset $\bar{L}$) if and only if $\exists y \in X$, such that, $y \notin \bigcup_{K=1}^{n} L^{(k)}$ and $(x, y) \in V$.

A vertex $x \in L^{(j)}$ is an ENP if and only if $\exists z \in X$ such that, $z \notin \bigcup_{K=1}^{n} L^{(k)}$ and $(z, x) \in V$. In the above definitions (x , y) is called an exit arc and (z, x) an entry arc .

2.3.9 Theorem 1 :- Let $L^{(r)} \in \bar{L}$ be a loop with an EXP x_{i_1} then $t(x_{i_1}) = \underline{P}$

Proof :- Let $L^{(r)}$ be the elementary circuit $\mu = x_{i_1}, x_{i_2} \cdots x_{i_k}, x_{i_l}$ & $\bar{L} = \{ L^{(1)}, L^{(2)} \cdots L^{(p)} \}$

$\because$ x_{i_1} is an $\exists$ X P $\exists y \in X$ such that $y \notin \bigcup_{r=1}^{p} L^{(r)}$ and $(x_{i_1}, y) \in V$ and $x_{i_1} \in L^{(r)}$.

$\because$ $x_{i_1} \in L^{(r)}$, $\exists z \in \mu$ such that $z \in \Gamma x \therefore \Gamma x = \{z, y\}$ (1

$\therefore$ $|\{\Gamma x_{i_1}\}| = 2$

$\therefore$ $t(x_{i_1}) = \underline{P}$ or CA (from the definition of G_Δ) (2

Suppose $t(x_{i_1}) = CA$ then let $\Gamma x_{i_1} = \{ x_{i_{l+1}}, x_p \}$

$\because$ $x_{i_1} \in \mu$, $x_{i_{l+1}}$ or $x_p \in \mu$

$\because z \in \Gamma x_{i_1}$ & $z \in \mu$ either $x_{i_{l+1}} = z$ or $x_p = z$

If $x_{i_{l+1}} \in \mu$ then $(t \neq x_p \in L^{(j)}, j \neq r, 1 \leq j \leq p$

If $x_p \in \mu$ then $x_{i_{l+1}} (= t) \in L^{(j)}, j \neq r, 1 \leq j \leq p$) (3)

From (1), (2) and (3) $\Gamma x_{i_1} = \{ z, y, t \}$

where $t \in$, X, $t = x_{i_{l+1}}$ or x_p , $t \in L^{(j)}, j \neq r, 1 \leq j \leq p$

$\because$ $|\{\Gamma x_{i_1}\}| = 3$

This is not possible in our flowgraph

$\therefore$ $t(x_{i_1}) \neq CA$. $\therefore$ $t(x_{i_1}) = \underline{P}$ QED

2.3.10 Let the loop $L^{(1)}$ be the elementary circuit $\mu_1 = x_{i_1}, x_{i_2}, \ldots x_{i_k}, x_{i_1}$ and the loop $L^{(2)}$ be the elementary circuit $\mu_2 = x_{j_1}, x_{j_2} \ldots x_{j_k}, x_{j_1}$. Let $S_1 = \{x \mid x \in \mu_1\}$ & $S_2 = \{x \mid x \in \mu_2\}$. The two loops are said to be <u>independent</u> if and only if $S_1 \cap S_2 = \emptyset$

The two loops are said to be <u>concentric</u> if either $S_1 = S_1 \cap S_2$ or $S_2 = S_1 \cap S_2$. In the former case $L^{(2)}$ is said to <u>enclose</u> or <u>contain</u> $L^{(1)}$ and in the latter vice-versa.

In all the other cases, the loops are said to be <u>intersecting</u>. Two loopsets are intersecting if any two loops - one from each loopset, intersect. A loop is said to be the <u>innermost</u> loop if it does not enclose any other loop.

2.3.11 <u>SIMPLE LOOP</u> :- A loop L is said to be simple if and only if

(1) L does not intersect any loop in the graph other than itself

(2) L does not enclose any other loop in the graph.

(3) L has exactly one EXP.

2.3.12 Let $\mu = x_{i_1}, x_{i_2} \ldots x_{i_k}$ be a path in G_Δ corresponding to a PS $\Delta(\alpha, \beta)$.

Let: $B(\mu) = \{v_i \mid \exists\ x \in \mu\}$ such that $t(x) = A$ and v_i is the assigned variable in this statement

$B(\mu)$ is called the set of all bound variables in μ

$F(\mu) = V - B(\mu)$ denotes the set of free variables in u where V is the set of all variables in PS $\Delta(\alpha, \beta)$

2.3.13 We now conclude this section after defining a function associated with the vertex x of a flowgraph G_Δ corresponding to the statement S in the PS $\Delta(\mathcal{X},\mathcal{B})$ for later use.

x = if S= $\left(\text{IF}(P)\uparrow_i\right)$ then P
else if S= $\left(\text{IF}(P)u = \sigma\right)$ then P
else if S= $\left(\text{READ } V_i\right)$ then V_i
else S.

—

§ 2.4 SEMANTICS OF EXECUTION OF A PS

Having described the syntax of a PS and its flowgraph we now consider the semantics of the PS. The various symbols and statements occuring in our PS have no semantic content assigned to them so far. An interpretation I is a domain D_I and a mapping of function and predicate symbols into functions and predicates and variable and constant symbols into constants in that domain D_I. I(l) stands for the interpretation assigned to the letter 'l'.

An interpreted PS is then, in effect, a computer program which could be executed on some idealized computer. In a given computational context, only a subclass of the set of all interpretations may be of interest. We define such a class of interpretations in a later chapter. Without loss of generality, we shall always consider the domain of interpretation to be the set of all positive integers N^+.

The exact number of variables occuring in a PS is of no importance, but only that their number should be finite, say ω. The values assigned to the variables in a PS are ordered in some arbitrary fashion and grouped in a vector η called the statevector, such that $\eta(v)$ gives the value associated with the variable 'v' if it has been assigned a value and is undefined otherwise. Hence we can think of it as the state of the computer executing the interpreted PS.

Let 'I' be an interpretation, z be a function term (2.2.1) and η a statevector. The semantics of the function term z with respect to I & η is given by

$$z(I,\eta) = \underline{\text{if}}\ z = v_i\ \ \underline{\text{then}}\ \eta(v_i)$$
$$\underline{\text{else if}}\ z = k_i\ \ \underline{\text{then}}\ I(k_i)$$
$$\underline{\text{else if}}\ z = f(\sigma \ldots \tau)\ \underline{\text{then}}\ I(f)(\sigma(I,\eta)\ldots\ (I,\eta)) \qquad \ldots.2.4.1.$$

where $\sigma, \ldots \tau$ are function terms.

The new state $(v_i = \sigma)\ (I,\eta)$ which results from the execution of the assignment $(v_i = \sigma)$ on the statevector η is computed by replacing the element in η corresponding to v_i

with $\tau(I, \eta)$. That is

$$(v_i = \tau)(I, \eta)(v_r) = \underline{\text{if}}\ r = i\ \underline{\text{then}}\ \tau(I, \eta)\ \underline{\text{else}}\ \eta(v_r)$$

for $1 \leq r \leq \omega$, where ω is the number of variables in PS .

....2.4.2.

The truth value of the simple predicate term (2.2.2.) $\rho_i = p_i^{(k_i)}(x_1, x_2 \ldots x_k)$ with respect to I and statevector η is obtained using the following

$$\rho_i(I, \eta) = \underline{\text{if}}\ I(p_i^{(k_i)})\left(\eta(x_1), \eta(x2) \ldots \eta(x_k)\right) = T\ \underline{\text{then}}\ T\ \underline{\text{else}}\ F$$

Where $\eta(x_i) = \underline{\text{if}}\ x_i \in V\ \underline{\text{then}}\ \eta(x_i)\ \underline{\text{else}}\ I(x_i)$.

The truth value of a predicate term P with respect to I and statevector η is obtained using the following.

$$P(I, \eta) = \underline{\text{if}}\ P = B(\rho_1, \rho_2 \ldots \rho_k)\ \underline{\text{then}}$$
$$\underline{\text{if}}\ B(\rho_1(I, \eta), \ldots \rho_k(I, \eta)) = T\ \underline{\text{then}}\ T$$
$$\underline{\text{else}}\ F \qquad \ldots (2.4.4.)$$

We are now in a position to describe computation in a PS, and for this purpose we institute a device like that instantaneous description of Turing machines. The output statevector resulting from the execution of an interpreted PS $\Delta(\alpha, \beta)$ with the flowgraph G_Δ, starting with an initial state vector η_I is denoted by $\Delta(I, \eta_I)$ and is computed by the partial execution function E.

that is $\Delta(I, \eta_I) = E(\Delta, I, \eta_I, x_I)$ where for any $x \in X$

$E(\Delta, I, \eta_I, x) =$ if $t(x) = A \wedge \Gamma_x = y$ then $E(\Delta, I, (x)(I, \eta_I), y)$

else if $t(x) = P$ or $CA_\wedge\Gamma_x = (y,z)$ then

if $(x)(I,\eta) = T$ then $E(\Delta I, \eta_I, y)$

else $E(\Delta, I, \eta_I, z)$

else if $t(x) = UT$ or $OP_\wedge\Gamma_x = y$ then $E(\Delta I, \eta_I, y)$

else if $t(x) = IP_\wedge\Gamma_x = y$ then $E(\Delta I, (x = A)(I, \eta_I), y)$

else if $t(x) = I_\wedge\Gamma_x = y$ then $E(\Delta, I, \eta_I, y)$

else if $t(x) = F$ then η_I 2.4.5

From the above it is clear that, if the statevector prior to executing the statement (in the PS $\Delta(\alpha,\beta)$ corresponding to the vertex x in G_Δ is η, then after execution of this statement the new statevector η_x is given by

$\eta_x =$ if $t(x) = A$ then $(x)(I, \eta)$

else if $t(x) = IP$ then $((x) = A)(I, \eta)$ else η

.... 2.4.6.

Hence, we see that the execution proceeds exactly as our intuition would indicate.

2.4.7 In the above the value of the partial execution function E at x, that is $E(\Delta, I, \eta, x)$, is equal to $E(\Delta, I, \eta_x, y)$ and is called the 4-tuple corresponding to the vertex x.

2.4.8 For a given PS $\Delta(\alpha,\beta)$ under some interpretation I with initial input statevector η_I, the sequence generated by the

partial execution function E acting on the flowgraph G_Δ is called the execution sequence EXS (Δ, I, η_I). The sequence is of the form

$$E(\Delta, I, \eta_I, x_I),\ E(\Delta, I, \eta_I x_{i_1}),\ E(\Delta I, \eta_{x_{i_1}}, x_{i_2}) \ \ldots\ldots$$
$$\ldots E(\Delta, I, \eta_{x_{i_j}}, x_{i_k}) \ldots$$

This implies the flow of control or the execution sequence traced the path $x_I, x_{i_1}, x_{i_2} \ldots x_{i_j}, x_{i_k} \ldots$ in G

2.4.9 Termination of a program schema :

A PS $\Delta(\alpha, \beta)$ is said to be terminating if and only if under any interpretation I and for any input state η_I - statevector η_I EXS (Δ, I, η_I) is finite.

2.4.10 Equivalence of two program schemas :

Two program schemas $\Delta(\alpha, \beta)$ & $\Delta(\alpha, \beta)$ are strongly equivalent if for every interpretation I and for the same set of values assigned to the input variables in α either both PS's terminate and yield the same set of values for the variables in β or both do not terminate.

2.4.11 Before we conclude this chapter we impose a a semantic condition on the flowgraphswhich we will consider, henceforth, for analysis. The condition is as follows.

For every vertex $x \in X$ and for all interpretations I of PS $\Delta(\alpha, \beta)$ there exists an input statevector η_I such that the path traced by $EXS(\Delta, I, \eta_I)$ meets the vertex x. In the above, η_I is called the <u>permissible statevector with respect to x</u>

....

CHAPTER 3

§ 3.1 INTRODUCTION

In this chapter we develop the condition for termination of interpreted program schemas. The model of computation developed in chapter 3 is used. The analysis is carried out at the metaprogram level which we assume, can be interpreted and executed by a computer. The symbols used in this analysis should not be confused with symbols in the ACL.

§ 3.2 Notations & Definitions

3.2.1. In this section we define some terms for later use.

It is easy to show that program schemas containing no loops always terminate. Unless otherwise mentioned the following set of notations will be used through the rest of this chapter.

Let $\Delta(x, B)$ be a P S with flowgraph G_Δ ;

* $L = x_{i_1}, x_{i2} \cdots x_{ik}, x_{i_1}$, be a loop in G_Δ ;

η_I be a permissible statevector (2.4.11) with respect to the BNP x_{im}; (x_{ij}, x_{in}) be an exit arc of L ;

B(L) be the set of bound variables in L denoted by $\{ v_j, v_{j2} \cdots v_{jm} \}$; In general, $B(\mu)$ denotes the set of bound variables in μ; V(L) be the set of all variables in L; The execution sequence E X S (Δ, I, η_I) traces an entry arc (z, x_{im}) of L, say. Then, there are two

---

* Henceforth, x_{it} stands for x_{i_t} where t is any string and i and x are any of the lower case letters.

consecutive 4-tuples of the form $E(\Delta, I, \eta_z, x_{im})$, $E(\Delta, I, \eta_z, x_{ip})$ in E X $S(\Delta, I, \eta_{i-1})$, where $(x_{im}, x_{ip}) \in L$ and η_z is statevector after executing the statement corresponding to the vertex z. η_z is called the input state vector of L with respect to ENP x_{im}

3.2.2 The execution sequence E X S (Δ, I, η_i) is said to exit from L thrd x_{ij} if it traces an exit arc (x_{ij}, x_{in}).

3.2.3 The loop L is said to be terminating if and only if for every interpretation of $\Delta(\alpha, \beta)$ and for every permissible statevector η_i with respect to any ENP of the loop L, the execution sequence EXS (Δ, I, η_{i-1}) exits from the loop.

3.2.4 The largest subsequence of 4-tuples of EXS (Δ, I, η_{i-1}) consisting of the 4-tuples corresponding to an ENP x_{im} of L upto and including the 4-tuple corresponding to an EXP x_{ij} of L constitutes the loop execution sequence LES (L, η_z, x_{ij}) of L. Note that the ENP of the loop is contained as a component in η_z.

3.2.5 If in the LES (L, η_z, x_{ij}) the 4-tuple corresponding to a vertex x occurs K + 1 times then the loop is said to have been executed k-times with respect to x .

3.2.6 TL $(\Delta, I, \eta_z, L, x_{ij}, x_{im}, i)$ stands for the statevector obtained after executing the loop L i-times

with respect to x_{ij} and with η_{iz} as the input statevector of L at the ENP x_{im}.

3.2.7 $T(\Delta, I, \eta_i, \mu, x_{j1})$ stands for the statevector obtained after executing the path $\mu = x_{j1}, x_{j2} \cdots x_{jk}$ in $\Delta(\alpha, \beta)$ under I and with a permissible statevector η_{i1} with respect to x_{j1}, where EXS (Δ, I, η_{i1}) trace (x, x_{j1})

3.2.8 Lemma:- $\forall$ I, EXS (Δ, I, η_i) exits from L thro' EXP x_{ij} if and only if the predicate Q defined below is true

Q = if $(x_{ij})\ (I, \eta_{x_{is}}) = T \wedge \psi(x_{ij}, x_{in}) = 1$

$\forall\ (x_{ij})\ (I, \eta_{x_{is}}) = F \wedge \psi(x_{ij}, x_{in}) = 0$

then T else F

where EXS (Δ, I, η_i) traces $(x_{is}, x_{ij}) \in L$. Proof for this lemma is immediate from (3.2.8)

3.2.9 Lemma:- A loop L terminates if and only if for every permissible statevector η_i with respect to any ENP of the loop L, the predicate Q defined as above is true. Proof follows from (3.2.8) & (3.2.4).

3.2.10 Loop predicate: If for a vertex $x \in V$, $t(x) = P$ & x is an EXP for some loop L then (x) is a loop predicate of L.

We assume every loop predicate is expressed in its d.n.f. In addition we assume the set of predicates is closed

under negation.

Therefore, given any PS $\Delta(\alpha, \beta)$ we can construct an equivalent PS $\Delta'(\alpha, \beta)$ such that in $G_{\Delta'} = (X', V', t, \Psi)$ if (x', y') is an exit arc of a loop L in $\Delta'(\alpha, \beta)$ then $\Psi(x', y') = 1$.

3.2.11 We note that the set of positive integers N^+ is well ordered with respect to $<$. Hence, there can not be an infinite decreasing sequence in N^+.

Bounded decreasing sequence

A sequence $x_1, x_2 \ldots x_k \ldots$ of integers in N^+ is a bounded decreasing sequence if for any i

$x_i > x_{i+1}$ and $x_i - x_{i+1} \leq x_i$

Limited Bounded Sequence is a bounded sequence ending in zero.

Predicate value sequence: If $p_j^{(k)}(x_1, x_2 \ldots x_k)$ is a k-ary simple predicate term in the d.n.f. of a loop predicate P corresponding to the EXP x_{ij} of L, then the sequence of q-tuple of values associated with the set of all variables in V(L), occurring in the sequence of statevectors

TL $(\Delta, I, \eta_z, L, x_{ij}, x_{im}, 1)$, TL $(\Delta, I, \eta_z, L, x_{ij}, x_{im}, 2) \ldots$ TL$(\Delta, I, \eta_z, L, x_{ij}, x_{im}, i) \ldots$ is called the predicate value sequence. The sequence is of the form $(v_j^{(1)}, v_{j2}^{(1)} \ldots, v_{jq}^{(1)})(v_j^{(2)}, v_{j2}^{(2)} \ldots v_{jq}^{(2)}) \ldots (v_j^{(i)}, v_{j2}^{(i)}, \ldots v_{jq}^{(i)})$ where $v_{j_r}^{(n)}$ denotes TL $(\Delta, I, \eta_z, L,$

x_{im}, n) (v_{jr}).

3.2.12 Slack function: Let Q be the predicate value sequence associated with a simple predicate term $T = p_j^{(k)}(x_1, x_2, \ldots, x_k)$ in the d.n.f. of the loop predicate P in the EXP x_{ij} of L. Then, a function $f : N^{+q} \to N$ is called a Slack function of the simple predicate term under the predicate value sequence Q if it satisfies either of the following conditions:

Condition 1a

$$\forall i,\ f(v_{j_1}^{(i)}, v_{j2}^{(i)} \ldots v_{jq}^{(i)}) > f(v_{j_1}^{(i+1)}, v_{j2}^{(i+1)} \ldots v_{jq}^{(i+1)})$$

$$\Rightarrow \exists_n \left(p_j^{(k)}(x_1^{(n)}, x_2^{(n)} \ldots x_k^{(n)}) \right) \wedge$$

$$(\forall_{r<n})(\sim p_j^{(k)}(x_1^{(r)}, x_2^{(r)} \ldots x_k^{(r)})$$

Or

Condition 1b

$$\forall i, f(v_{j_1}^{(i)}, v_{j2}^{(i)} \ldots v_{jq}^{(i)}) > f(v_{j_1}^{(i+1)}, v_{j2}^{(i+1)} \ldots v_{jq}^{(i+1)})$$

$$\Rightarrow (\exists_n)\left(p_j^{(k)}(x_1^{(n)}, x_2^{(n)}, x_k^{(n)}) \right) \wedge$$

$$(\forall_{r<n})\left(\sim p_j^{(k)}(x_1^{(r)}, x_2^{(r)} \ldots x_k^{(r)}) \right) \wedge$$

$$f(v_{j_1}^{(i)}, v_{j2}^{(i)} \ldots v_{jq}^{(i)}) - f(v_{j_1}^{(i+1)}, v_{j2}^{(i+1)} \ldots v_{jq}^{(i+1)})$$

$$\leq f(v_{j_1}^{(i)}, v_{j2}^{(i)}, \ldots v_{jq}^{(i)})$$

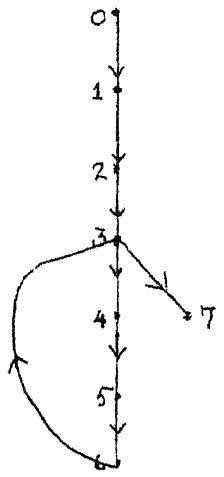
3.2.13 Allowable interpretation:- An interpretation I is said to be allowable for $\Delta(\alpha, \beta)$ if there exists a slack function associated with atleast one of the simple predicates terms in the d.n.f.of at least one of the loop

predicates of every loop L in G_Δ .

Intuitively this means that we allow only those interpretations of a program schema in which with some loop predicates of every loop, we can associate a slack function that is monotonically decreasing or monotonically decreasing in a bounded fashion. Hence, the basic philosophy of our approach to prove termination is to construct a weight function for every loop for the given interpretation. Success in such an attempt for any loop predicate implies the loop terminates because of condition 1 of lemma 4.2.9. We exemplify our technique with the following example.

Example 1 Let us consider the following interpreted program schema with the standard interpretation of the symbols:

1. i = 1
2. S = 0
3. IF (i < n) ↑7
4. S = S + A
5. i = i + 1
6. ↑ 3
7. STOP



Let α =(n,A), β=(i,s)

PS is $\Delta(\alpha, \beta)$

G_Δ is given above

$L = x_3\ x_4\ x_5\ x_6\ x_3$ is a loop in G_Δ with $i > n$ as the loop predicate. Let us define n-i as the slack function associated with this loop predicate. Hence, for every execution of the loop n-i decrease and once it become negative the execution sequence exits from the loop and hence our usual interpretation of $>$ is an allowable interpretation. This is obviously true for any $x \in N^+$. Hence, we conclude that the loop terminates and hence the program terminates.

We note that the loop predicate is satisfied for the first time the value of the slack function is negative. It need not always be true for all allowable interpretation that the loop predicate becomes true only when the slack function becomes negative.

§ 3.3 A Heuristic Algorithm For Finding A Slack Function.

3.3.1 In this and the next section we develop an heuristic algorithm for finding a slack function, in a real life PS in which the predicate symbols allowed are $<, \leq, =, \neq, >, \& \geq$ and the function symbols are +, - , *, & / . We will assign these function and predicate symbols only the usual interpretation given to them in arithmatic. Other symbols are as in the ACL. We assume, unless otherwise mentioned, that all the PS we consider for analysis in this section and further have no intersecting loopsets in their flowgraphs.

3.3.2 Let $\mu = x_{i1}, x_{i2} \ldots x_{ik}$ be a path in a PS $\Delta(\alpha, \beta)$.
Let $B(\mu) = \{ v_{j1}, v_{j2}, \ldots v_{jn} \}$

the free computation sequence of the path μ, $Z(\mu)$, is a sequence of n-tuple of strings of symbols defined inductively as follows. ($Z(\mu)(x)$ stands for the term in $Z(\mu)$ corresponding to the vertex $x \in \mu$, and $Z(\mu)(x)(v_{ji})$ stands for the i^{th} element or the element corresponding to v_{ji} in this term). That is, $Z(\mu) = Z(\mu)(x_{i1}), Z(\mu)(x_{i2}), \ldots Z(\mu)(x_{ik})$, where

(1) $Z(\mu)(x_{i1}) =$ if $t(x_{i1}) = A \wedge (x_{i1}) = (v_{jr} = \sigma)$ then

$\langle v_{j1}, v_{j2}, \ldots v_{jr-1}, \sigma, v_{jr+1}, \ldots v_{jn} \rangle$

else if $t(x_{i1}) = IP \wedge (x_{i1}) = v_{jr}$ then

$\langle v_{j1}, v_{j2}, \ldots v_{jr-1}, C, v_{jr+1}, \ldots v_{jn} \rangle$

else $\langle v_{j1}, v_{j2}, \ldots v_{jn} \rangle$, where $C \in N^+$

(2) for $1 < j \leqslant k-1$

$Z(\mu)(x_{ij+1}) =$ **if** $t(x_{ij+1}) = A \wedge (x_{ij+1}) = (v_{jr} = \sigma^{-})$ **then**
$\langle Z(\mu)(x_{ij})(v_{j1}), \ldots Z(\mu)(x_{ij})(v_{jr-1}), \sigma^{-}{}^{w_{j1}, w_{j2}, \ldots w_{jt}}_{\alpha_{r1}, \alpha_{r2}, \ldots \alpha_{rt}}, \ldots Z(\mu)(x_{ij})(v_{jn}) \rangle$

else if $t(x_{ij+1}) = IP \wedge (x_{ij+1}) = v_{jr}$ **then**
$\langle Z(\mu)(x_{ij})(v_{j1}), \ldots Z(\mu)(x_{ij})(v_{jr-1}), 0, \ldots Z(\mu)(x_{ij})(v_{jn}) \rangle$
else $Z(\mu)(x_{ij})$

where w_{j1}, w_{j2}, w_{jt} are the variables occuring in σ^{-}
and $\alpha_{rk} =$ **if** $w_{jk} = v_{js}$, $1 \leqslant s \leqslant n$ **then** $Z(\mu)(x_{ij})(v_{js})$
else w_{jk}

and $S^{t_1, t_2, \ldots t_n}_{q_1, q_2, \ldots q_n}$ stands for the string obtained by syntactic substitution of $q_1, q_2, \ldots q_n$ for $t_1, t_2, \ldots t_n$ respectively in S, and $t_1, t_2, \ldots t_n, q_1, q_2, \ldots q_n$ are strings of symbols.

$Z(\mu)(x_{ik})$ is called the result* of the path μ, and is denoted by $a(\mu)$ & $h(\mu)(v_{ji})$ denotes $Z(\mu)(x_{ik})(v_{ji})$.

3.3.3 Lemma:- Let $\mu = x_{i1}, x_{i2}, \ldots x_{ik}$ be a path and $B(\mu) = v_{j1}, v_{j2}, \ldots v_{jn}$ be the set of bound variables in μ. Then, for $1 \leqslant i \leqslant n$ $R(\mu)(v_{ji})$ is a function term.

PROOF:- Proof is by induction on the number of vertices in μ.

Basis - Consider the path $\mu' = x_{i1}$ with one vertex.

* The computation path may also be henceforth, called as the free computation vector.

$\therefore$ $R(\mu') = Z(\mu')(x_{i1}) =$ if $t(x_{i1})=A$ $(x_{i1})=(v_{jr}=\sigma)$ then

$\langle v_{j1}, v_{j2}, \ldots v_{jr-1}, \sigma, v_{jr+1}, \ldots v_{jn} \rangle$

else if $t(x_{i1})=IP \wedge (x_{i1})=(v_{jr})$ then

$\langle v_{j1}, v_{j2}, \ldots v_{jr-1}, C, v_{jr+1}, \ldots v_{jn} \rangle$

else $\langle v_{j1}, v_{j2}, \ldots v_{jn} \rangle$

where σ is a function term, C is a constant.

$\therefore$ $R(\mu')(v_{ji})$ for $1 \leqslant i \leqslant n$ is a function term.

Induction hypothesis:- Let $\mu'' = x_{i1}, x_{i2}, \ldots x_{im-1}$ be a path with $(v_{j1}, v_{j2}, \ldots v_{jq})$ as the set of bound variables in it. Then, $R(\mu'')(v_{ji})$, $1 \leqslant i \leqslant q$ is a function term.

We have to show that for $\mu''' = x_{i1}, x_{i2}, \ldots x_{im-1}, x_{im}$, $R(\mu''')(v_{ji})$ is a function term.

But $R(\mu'')(v_{ji}) = Z(\mu''')(x_{im-1})(v_{ji})$, $1 \leqslant i \leqslant q$.

Case 1:- Let $t(x_{im}) = A$ and $(x_{im})=(v_{jr}=\sigma)$, $1 \leqslant r \leqslant q$ and v_{jr} is bound in μ'''.

Then, $Z(\mu''')(x_{im}) = \langle Z(\mu''')(x_{im-1})(v_{j1}), \ldots Z(\mu''')(x_{im-1})(v_{jr-1}), \sigma \begin{matrix} w_{j1}, w_{j2}, \ldots w_{jt} \\ \alpha_{r1}, \alpha_{r2}, \ldots \alpha_{rt}, \end{matrix} \ldots Z(\mu''')(x_{im-1})(v_{jq}) \rangle$ ...(1)

$= \langle Z(\mu'')(x_{im-1})(v_{j1}), \ldots Z(\mu'')(x_{im-1})(v_{jr-1}) \sigma \begin{matrix} w_{j1}, w_{j2}, \ldots w_{jt} \\ x_{r1}, x_{r2}, \ldots x_{rt} \end{matrix} , \ldots Z(\mu'')(x_{im-1})(v_{jq}) \rangle$ ...(2)

$= \langle R(\mu'')(v_{j1}), \ldots R(\mu'')(v_{jr-1}), \sigma \begin{matrix} w_{j1}, w_{j2}, \ldots w_{jt} \\ \alpha_{r1}, \alpha_{r2}, \ldots \alpha_{rt}, \ldots \end{matrix}$

$\ldots R(\mu'')(v_{jq}) >$

where α_{jk} = if $w_{jk} = v_{js}$ $1 \leq s \leq q$ then $Z(\mu''')(x_{im-1})(v_{js})$
else w_{jk}.

= if $w_{jk}=v_{js}$ $1 \leq s \leq q$ then $Z(\mu'')(x_{im-1})(v_{js})$
else w_{jk}.

= if $w_{jk}=v_{js}$ $1 \leq s \leq q$ then $R(\mu'')(v_{js})$
else w_{jk}.

But by [illegible] $R(\mu'')(v_{ji})$ is a function term, and w_{jk} is a function term.

$\therefore \sigma$ $\begin{matrix} w_{j1}, w_{j2} \ldots w_{jt} \\ x_{r1}, x_{r2} \ldots x_{rt} \end{matrix}$ is a function term

Case 2 - let $t(x_{jm}) = A$ and $(x_{im})=(v_{jr}= \sigma)$ and $v_{jr} \notin B(\mu'')$ then obviously $R(\mu''')(v_{jr})$ is σ that is, a function term

$\therefore R(\mu''')(v_{jr})$, $1 \leq r \leq q$ is a function term.

Case 3:- Similarly, it is seen that in all other cases, that is, $t(x_{jm}) \neq A$, $R(u''')(v_{ji})$ is a function term. QED

Hence, we note that function terms are closed under the operation of syntactic substitution of function terms.

3.3.4 Theorem:- Let $\mu = x_{i1}, x_{i2}, \ldots x_{ik}$ be a path in a PS $\Delta(\alpha,\beta)$ and $(v_{j1}, v_{j2}, \ldots v_{jn})$ be the bound variables in it then,

$\forall i,\ 1 \leqslant i \leqslant n, (T(\Delta, I, \eta_x, \mu, x_{i1}))(v_{ji}) = R(\mu)(v_{ji})(I, \eta_x)$

where for η_I a permissible statevector w.r.t x_{i1} the TXS(Δ, I, η_I) traces (x, x_{i1})

($T(\Delta I, \eta_x, \mu, x_{i1})$ is abbreviated as $T'(\eta_x, \mu, x_{i1})$)

PROOF:- We induct on the number of vertices in the path μ.

Basis:- Let $\mu' = x_{i1}$, be a path with one vertex.

If $t(x_{i1}) = \Lambda$ and $(x_{i1}) = (v_{jr} = \sigma)$ then $R(\mu') = \langle v_{j1}, v_{j2}, \ldots \sigma, \ldots v_{jn} \rangle$

$\therefore \forall i,\ 1 \leqslant i \leqslant n,\ R(\mu')(v_{ji})(I, \eta_x) =$ if $i \neq r$ then $\eta_x(v_{ji})$ else $\sigma(I, \eta_x)$

From the definition of T we have $T'(\eta_x, \mu', x_{i1}) = \eta_{x_{i1}}$

Then, $\eta_{x_{i1}}(v_{ji}) = (v_{jr} = \sigma)(I, \eta_x)(v_{ji}) =$ if $i=r$ then $\sigma(I, \eta_x)$ else $\eta_x(v_{ji})$

$\therefore \forall i,\ 1 \leqslant i \leqslant n,\ T'(\eta_x, \mu', x_{i1})(v_{ji}) = R(\mu')(v_{ji})(I, \eta_x)$

Now, if $t(x_{i1}) \neq \Lambda$ then it immediately follows that the theorem is true for a path consisting of the single vertex.

Induction Hypothesis:-

$\forall i,\ 1 \leqslant i \leqslant n$, $(T'(\eta_x, \mu'', x_{i1}))\ (v_{ji}) = R(\mu'')(v_{ji})(I, \eta_x)$

where $\mu'' = x_{i1}, x_{i2}, \ldots x_{im-1}$

Let $\mu''' = x_{i1}, x_{i2}, \ldots x_{im-1}, x_{im}$

$$\therefore x_{rk}(I,\eta_x) = R(\mu'')(v_{js})(I,\eta_x)$$
$$= (T'(\eta_x,\mu'', x_{i1}))(v_{js}) \quad \text{by I.H.}$$
$$= \eta_{x_{im-1}}(v_{js})$$

OR

(2) If $w_{jk} \notin B(\mu''')$ then $x_{rk} = w_{jk} = v_{jp}$

$$\therefore \alpha_{rk}(I,\eta_x) = w_{jk}(I,\eta_x)$$
$$\because w_{jk} \notin B(\mu''')$$
$$w_{jk}(I,\eta_x) = \eta_{x_{im-1}}(v_{jp})$$

In $\sigma^{w_{j1}, w_{j2}, \cdots w_{jt}}_{\alpha_{r1}, \alpha_{r2}, \cdots \alpha_{rt}}$ every variable in σ is substituted by the corresponding α_r and the value assigned to these α_r's is the value that the corresponding variable α_r replaces, has in $\eta_{x_{im-1}}$. That is, every variable in σ is assigned the value it would have in $\eta_{x_{im-1}}$

$$\therefore \sigma^{w_{j1}, w_{j2}, \cdots w_{jt}}_{\alpha_{r1}, \alpha_{r2}, \cdots \alpha_{rt}}(I,\eta_x) = \sigma(I,\eta_{x_{im-1}})$$

$$\therefore R(\mu''')(v_{jr})(I,\eta_x) = \sigma(I,\eta_{x_{im-1}})$$

Also, $(T'(\eta_x, \mu''', x_{i1}))(v_{jr}) = \eta_{x_{im}}(v_{jr})$

$$= ((v_{jr} = \sigma)(I,\eta_{x_{im-1}}))(v_{jr})$$
$$= \sigma(I, \eta_{x_{im-1}}) \qquad \text{QED}$$

From this it is seen that the component corresponding to a variable in the result of a path performs the same transformation of its value as the path would do for the same input vector.

3.3.5 COROLLARY :- Let $\mu = x_{i1}, x_{i2}, \ldots x_{ik}$ be a path in the flowgraph G_Δ of a PS$\Delta(\alpha,\beta)$. Let $t(x_{ik}) = \underline{P}$ or CA, η_I be a permissible statevector w.r.t. x_{i1} and I be any interpretation. Suppose EXS (Δ, I, η_I) traces the input arc (x, x_{i1}) and let η_x be the statevector after executing the statement corresponding to vertex x. Then,

$$(x_{ik})(I,\eta_{x_{ik}})=T \iff (x_{ik})\begin{matrix} w_{j1}, w_{j2}, \ldots w_{jt} \\ \alpha_{r1}, \alpha_{r2}, \ldots \alpha_{rt} \end{matrix} (I,\eta_x)=T$$

$w_{j1} \ldots . w_{jt}$ are all the variables occuring in the predicate (x_{ik}) and $x_{r1} \ldots x_{rt}$ are the components of $R(\mu)$ corresponding to $w_{j1}, \ldots w_{jt}$

PROOF:- From 3.3.4 we have

$\forall i,\ 1 \le i \le n,\ Z(\mu)(x_{ik-1})(v_{ji})(I,\eta_x)=(\underline{T}'(\eta_x,\mu'',x_{i1}))(v_{ji})(I,\eta_x)$

where $\mu'' = x_{i1}, x_{i2}, \ldots x_{ik-1}$

If $w_{ji} = v_{jq}$ then from 3.3.4 $x_{ji}(I,\eta_x) = \eta_{x_{ik-1}}(v_{jq})$

and $\because$ $t(x_{ik}) = \underline{P}$ or CA and $\eta_{x_{ik-1}} = \eta_{x_{ik}}$

$$\therefore (x_{ik})\begin{matrix} w_{j1}, w_{j2}, \ldots w_{js} \\ x_{j1}, x_{j2}, \ldots x_{js} \end{matrix} (I,\eta_x)=(x_{ik})(I,\eta_{x_{ik}}) \qquad \text{QED}$$

3.3.6 Having developed the necessary results, we now present our heuristic algorithm for obtaining a slack function.

Let $L = x_{i_1}, x_{i2}, \cdots x_{ik}$ be a simple loop. We assume for simplicity, that the loop predicate at the EXP x_{ij} is a simple predicate term. Let $(x_{ij}) = w_1 p w_2$ where $p \in \{<, \leq, =, \neq, >, \geq\}$ and $w_1, w_2 \in \{\mathcal{V} \cup C\}$; $\mu = x_{ij+1}, \cdots x_{ik}, x_{i1}, \cdots x_{i1}, \cdots x_{ij-1}, x_{ij}$, η_L be the input statevector of L w.r.t an ENP x_{im}; (x, z) be an exit arc, $B(\mu)$ is the set of bound variables in μ, $(v_{j1}, v_{j2}, \cdots v_{jn})$; η_I be the permissible statevector w.r.t. ENP x_{im}.

The path traced by the EXS(Δ, I, η_I), if the loop were executed r - times w.r.t. EXP x_{ij}, would be of the form $\mu' = x_{im}, x_{im+1} \cdots x_{ij} (\mu)^r$. It is clear from the above that the loop L terminates (exits thro' x_{ij} always) if and only if the loop $L' = x_{ij+1}, \cdots x_{ik}, x_{i1} \cdots x_{ij-1}, x_{ij}, x_{ij+1}$ terminates.

Let:

$M(w_1, w_2)$ = if p is $> \vee \geq$ and $\psi(x, z) = 1$, p is $< \vee \leq$ and $\psi(x,z) = 0$ then $w_2 - w_1$

else if p is $> \vee \geq \wedge \psi(x,z) = 0 \vee$ p is $< \vee \leq \wedge \psi(x,z) = 1$ then $w_1 - w_2$

else $(w_1 - w_2)$ or $(w_2 - w_1)$

Let:

α_r^n , $1 \leq r \leq 2$ = if $w_r = v_{qs}$ and $v_{qs} \in B(u)$
then $R(\mu^n)(v_{qs})$ else w_r
where μ^n denotes the path repeated r -times.

Let:

$f(w_1^{(n)}, w_2^{(n)}) = f_n(w_1, w_2) = M^{w_1, w_2}_{\alpha_1^n, \alpha_2^n}$

$f(w_1^{(n+1)}, w_2^{(n+1)}) = f_{n+1}(w_1, w_2) = M^{w_1, w_2}_{\alpha_1^{n+1}, \alpha_2^{n+1}}$

where $w_i^{(n)}$ denotes the valu w_i takes in the loop at the end of n - iterations.

We know from (3.3.4) for $1 \leq s \leq n$, TL $(\Delta, I, n_L, L, x_{ij}, x_{im}, n)(v_{ji}) = R(\mu^n)(v_{ji})(I, n_L)$ .....(1)

& TL $(\Delta, I, n_L, L, x_{ij}, x_{im}, n+1)(v_{ji}) = R(\mu^{n+1})(v_{ji})(I, n_L)$ .....(2)

we also have from (3.3.5)

$(x_{im})^{w_1, w_2}_{\alpha_1^n, \alpha_2^n}(I, n_L) = T \Leftrightarrow (x_{im})(I, TL(\Delta, I, n_L, x_{ij}, x_{im}, n) = T$

& $(x_{im})^{w_1, w_2}_{\alpha_1^{n+1}, \alpha_2^{n+1}}(I, n_L) = T \Leftrightarrow (x_{im})(I, TL(\Delta, I, n_L, x_{ij}, x_{im}, n+1)$

for $1 \leq i \leq 2$ such that $w_i \in B(u)$ we have from (1)

TL $(\Delta, I, n_L, L, x_{ij}, x_{im}, n)(w_i) = R(\mu^n)(w_i)(I, n_L)$

$\therefore f_n(w_1, w_2) = M(I, TL(\Delta, I, n_L, L, x_{ij}, x_{im}, n)$

$= M^{w_1, w_2}_{\alpha_1^n, \alpha_2^n}(I, n_L)$

$\therefore f_n(w_1, w_2) - f_n(w_1, w_2)$

$= (M^{w_1, w_2}_{\alpha_1^n, \alpha_2^n} - M^{w_1, w_2}_{\alpha_1^{n+1}, \alpha_2^{n+1}})(I, n_L)$. Now

it is clear from the construction that $\forall n, f_n(w_1,w_2) \geq f_{n+1}(w_1,w_2)$

$\Rightarrow \exists n(p^2(w_1^{(n)}, w_2^{(n)})) \wedge (\forall s < n)(\neg p^2(w_1^{(s)}, w_2^{(s)}))$. But

If p is = and $\psi(x,z) = 1$ or p is $\neq$ and $\psi(x,z) = 0$ then we have to show that $f_n(w_1, w_2) - f_{n+1} \leq f_n(w_1, w_2)$

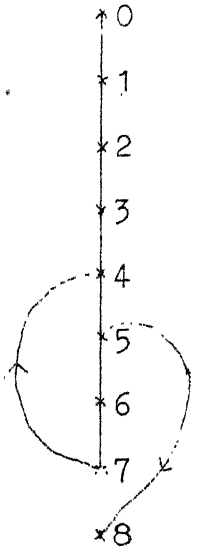
$\therefore$ We will be thro' if we show that V

$\forall n, f_n(w_1, w_2) > f_{n+1}(w_1, w_2)$.

For this purpose we give here some examples using standard symbols manipulation techniques. Some more complicated ones are done in the appendix.

EXAMPLE 1.- Consider the PS $\Delta(\alpha, \beta)$ given below with standard interpretations for the symbols.

1. READ Y
2. READ Z
3. x = 0
4. x = x+z
5. IF(x y) 8
6. y = y-1
7. 4
8. STOP



The flowgraph G_Δ is given above.

$L = 4,5,6,7,4; \quad \mu = 6,7,4,5; \; w_1 = x; \; w_2 = y$

Let:

$$R(\mu^o) = (X_o, Y_o, Z_o)$$

Assume as induction hypothesis -

$$R(\mu^n) = (X_o + nZ_o, Y_o - n, Z_o)$$

$$\therefore Z(\mu^{n+1})(6) = (X_o + nZ_o, y_o - (n+1), Z_o)$$

$$Z(\mu^{n+1})(7) = (X_o + nZ_o, y_o (n+1), Z_o)$$

$$Z(\mu^{n+1})(4) = (X_o + nZ_o, y_o(n+1), Z_o)$$

$$\therefore R(\mu^{n+1}) = (X_o + (n+1)Z_o, y_o - (n+1), Z_o)$$

$$\therefore f_n(w_1, w_2) = M^{w_1, w_2}_{\alpha_1^n, \alpha_2^n} = (y_o - n) - (X_o + nZ_o) \quad \text{and}$$

$$\therefore f_{n+1}(w_1, w_2) = M^{w_1, w_2}_{\alpha_1^{n+1}, \alpha_2^{n+1}} = (y_o - (n+1) - (X_o(n+1) Z_o)$$

$$\therefore f_n - f_{n+1} = Z_o + 1 > 0$$

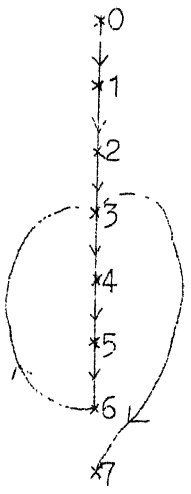
since at any stage in our computation every variable has been assumed to take only positive integral values. It is clear that

$$\forall n, \; f_n > f_{n+1}$$

$\therefore$ the program terminates.

EXAMPLE 2: Consider the PS give below

1. READ x
2. READ y
3. IF ($x \geqslant y$) $\uparrow_7$
4. y = y-x
5. x = x+y
6. $\uparrow_3$
7. STOP



The flowgraph is givenabove.

$w_1 = x$; $w_2 = y$

L = 3,4,5,6,3; μ = 4,5,6,3,

Let:

$$R(\mu^n) = (\alpha_1^n , \alpha_2^n)$$

$$\therefore \quad R(\mu^{n+1}) = (\alpha_1^n + \alpha_2^{n+1}, \alpha_2^{n+1} = \alpha_2^n - \alpha_1^n)$$

$$= (\alpha_2^n , \alpha_2^n - \alpha_1^n)$$

$f_n(w_1 , w_2) = \alpha_2^n - \alpha_1^n$ and

$f_{n+1}(w_1, w_2) = \alpha_2^n - \alpha_1^n - \alpha_2^n = - \alpha_1^n$

$\therefore f_n - f_{n+1} = \alpha_2^n$

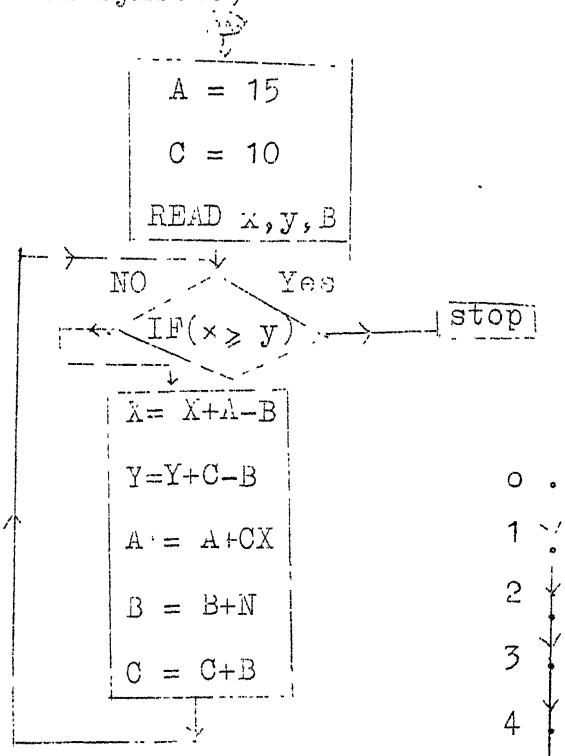
But from (1) above we know that

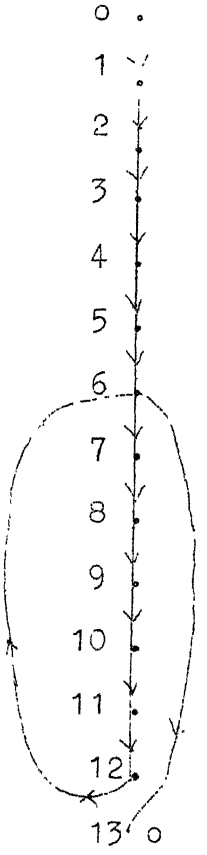
$TL(\wedge I, n_{ij}, L, x_{ij}, x_{im}, n)(v_{ji}) = \alpha_2^n \ (I, n_{ij})$

$\therefore \forall n, f_n - f_{n+1} > 0$; hence, program terminates since every variable takes only positive values.

EXAMPLE 3: Consider the PS given below with standard interpretations for symbols,

1. A= 15
2. C= 10
3. READ x
4. READ y
5. READ B
6. IF (x ⩾ y) ↑13
7. X = X+A-B
8. Y = Y+C-B
9. A = A+CX
10. B = B + N
11. C = C + B
12. ↑6
13. STOP





The flowgraph is given above

L = 6,7,8,9,10,11,12,6

μ = 7,8,9,10,11,12, 6

$w_1 = x$; $w_2 = y$

Let:

$R(\mu^o) = (X_o, Y_o, A_o, B_o, C_o)$

$R(\mu^n) = (X_n, Y_n, A_n, B_n, C_n)$

$R(\mu^{n+1}) = (X_n+A_n - Bn, Y_n+ C_n - B_n, A_n + C_nX_{n+1}, B_n+N, C_n+B_{n+1}$

$\therefore f_n(w_1\ w_2) = Y_n - X_n$

$\therefore f_n(w_1 , w_2) = Y_{n+1} - X_{n+1}$

$$\therefore f_n(w_1, w_2) - f_{n+1}(w_1, w_2)$$
$$= (Y_n - X_n) - (Y_{n+1} - X_{n+1})$$
$$= (Y_n - X_n) - (Y_n + C_n - X_n - A_n)$$
$$= A_n - C_n$$

Now, if $A_n > C_n$ for all then $\forall n$, $f_n > f_{n+1}$

Let: assume $A_o > C_o$ (in this particular example this is true

Let us assume as Induction Hypothesis is A_{n-1} C_{n-1}

It is clear that

$$A_n = A_{n-1} + C_{n-1} X_n$$

$$C_n = C_{n-1} + B_n$$

$$\therefore A_n - C_n = (A_{n-1} - C_{n-1}) + (C_{n-1}X_n - B_n)$$

$\therefore$ From IH it is clear that $A_n > C_n$ if and only if

$$C_{n-1}X_n - B_n > o$$

It is easily seen that

$$B_n = B_o + nN$$

Now, $C_{n-1} = C_{n-2} + B_{n-1}$

$$= C_{n-2} + (B_o + (n-1)N)$$
$$= C_{n-3} + 2B_o + N((n-1) + (n-2))$$
$$= C_o + (n-1)B_o + N\left\{\sum_{i=1}^{n-1}(n-i)\right\}$$

$$= C_o + nB_o + n^2N + D$$

where D is a constant

Now, since $X_n(I, n_L) = TL(\Delta I, n_L, L, 6, 6, n)$ (x)

it is clear that the value of $X_n > 0$ because no variable in our program takes a negative values.

$\therefore\ C_{n-1} * X_n = C_o X_n + nB_o X_n + n^2 N X_n + D X_n$

and $B_n = B_o + nN$

It follows that $C_{n-1} X_n > B_n$. Hence the program terminates.

In some simple enough cases such as Example 1 above it is relatively easy to get an induction hypothesis concerning the form of the result or free computation vector at the end of n^{th} iteration of the loop. It is of course, then easy to verify this assumption by one more iteration. However, in some cases, such as Example 2 above, it is not easily formed. In such cases, we use subscripted variable symbols to denote the corresponding component of the free computation vector after n - iterations. The proof of termination then proceeds in terms of these symbols. In still other cases, we have to develop the explicit expression for some components of the free computation vector. In example 3 above, we have had to do this for the program variables B and C.

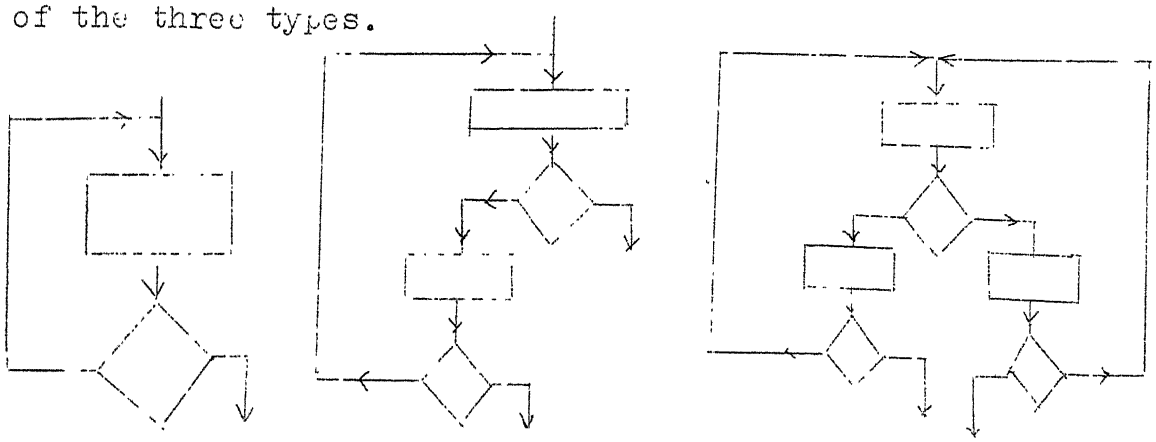
As illustrated in the above example, we have used a somewhat informal method for proving program termination. In the first place the construction of the slack function for the standard relational predicates, has been made in such a way as to satisfy condition 1a. Secondly, in showing that the slack function takes decreasing values for any two successive iterations of the loop, we have made use of properties of program variables that are intuitively true. The entire formalism could have been developed in terms

of a standard logical system, such as the K - system of Mendelson, suitably modified. If the axiom schemata in such a system is augmented by axion defining the slack function for a particular predicate in the system then condition 1 would be a provable theorem. Formalisation of the proofs we have given above immediately yields the desired result by modus ponens. We do not propose to carry out this formalisation in any greater detail as we are convinced that very little is to be gained by this.

Proving that the slack function is monotonically decreasing for any two consecutive iterations of the loop is quite often a relatively simple task even for fairly large programs. This is so because explicit recognition is taken of the dependence of the program variables on each other. Thus, assignments which in no way affect the truth or falsity of the loop predicate can be ignored. This is in contrast to the method developed by Z.Manna (23) in which case the size of the wff associated with the program is directly proportional to the size of the program.

Now, if the loop predicate contains more than one simple predicate term in its d.n.f. then the loop is guaranteed to terminate if for any of the simple predicate terms the slack function satisfies condition 1. We also note, using the result of Bohm & Jacopini (41), that any loop can be put in one

of the three types.



Type 1 Type 2. Type 3

We have given an algorithm to construct a slack function for type 1 innermost loops. Now, we note that if in a type 2 innermost loop, the function found using the above algorithm for at least one simple predicate term in the d.n.f. of at least one of the loop predicates, ignoring others satisfies the condition 1, then the loop terminates.

§ 3.4 PERSISTENCE

We consider the termination of a loopset in this section.

3.4.1 Let $\bar{L} = (L^{(1)}, L^{(2)}, \ldots L^{(n)})$ be a loopset in a PS $\Delta(x, \beta)$ with a flowgraph G_Δ ; I be any interpretation and n_I be a permissible statevector with respect to an ENP of the loop $L^{(r)}$, $1 \leq r \leq n$. .

We say an EXS(Δ, I, n_I) enters the loop $L^{(r)}$ if the EXS (Δ, I, n_I) traces an arc $(x, y) \in L^{(r)}$

such that $\forall j, 1 \leq j \leq n, j \neq r, (x,y) \notin L^{(j)}$

$\bar{L}$ is said to be persistent if ENS(Δ, I, n_I) enters the loop $L^{(r)}$ then it does not enter any other loop before it exits, if at all, from the loopset.

Note that the property of persistence is for all values of the permissible statevector w.r.t. every ENP of the loop.

A simple of a persistent loopset is one in which every predicate which is not a loop predicate, the variables occuring in it are free in that loop.

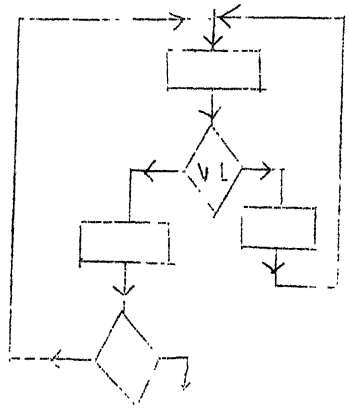
3.4.2. Theorem: Let $\bar{L}$ be a persistent loopset and let every loop in it terminate. Then, the loopset terminates.

Proof for this theorem is immediate from the definition of persistence.

Consequence of the persistence property of a loopset is that we can consider the different loops in the loopset independently of each other for proving termination and conclude the termination of the entire loopset.

§ 3.5 REMARKS AND CONCLUSIONS

We notice that if every loopset in the flowgraph is persistent then the above technique can obviously be applied even to flowgraphs containing intersecting loopsets. Now, if a loop in a loopset has no EXP as below then, we notice that by defining V1 to be the EXP of L, we can apply the given technique. The given algorithm can be easily extended to cases where loopsets may not be persistent but the pattern of alternation between the various loops in the loopset is known in advance.



We have not included "DO" type statements as in FORTRAN in our ACL because "DO" loops with no conditional loops always terminate. Our algorithm can be easily be extended to deal with loops containing "DO" loops. We have not allowed subscripted variables in our ACL because it does not have "DO" type statements in it. It is easy to see that the computational power of ACL is not hampered by not including these features.

It was found by examining about sixty programs (30 short jobs, 20 express jobs, 10 long jobs) submitted at the Computer Centre in I.I.T., Kanpur that very few (4) programs had any

concentric loops in them. There was not even a single program having a loopset of the type 3a. The loopsets were found to be almost always persistent and even for those minority of cases where loopsets were not persistent the pattern of alternation between the loops of the loopset could be very easily found. The algorithm was worked out manually with great success, suing only very obvious facts, in all but two cases.

In the Appendix we have worked out our algorithm for some interesting cases.

-oOo-

CHAPTER 4

§ 4.1 Suggestions

The algorithm outlined has been found to work successfully in most of the programs taken for manual check through, without the need for very complex heuristics.

One very simple case for which the algorithm given here fails is given below (30).

Example: (Euclid's algorithm): Given two positive integers m and n find their greatest common divisor i.e. the largest positive integers which evenly divides both m and n.

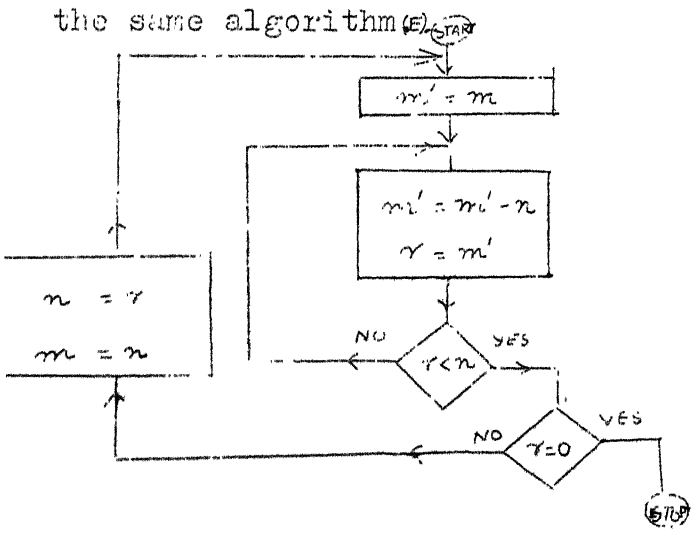
step 1 m^1 = m is the larger of the two numbers.

Step 2 Find the remainder

Step 3 If the remainder is zero then the algorithm terminates with n as the answer

Step 4 m = n; n = r go to step 1

It is easily seen that the flowchart given below carries out the same algorithm

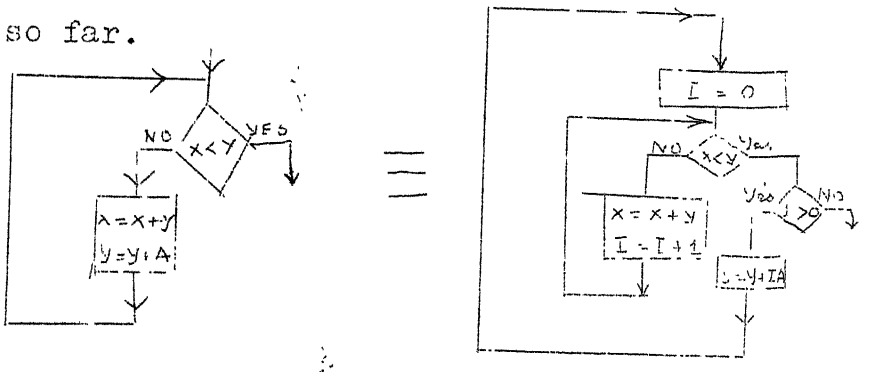


The algorithm E can be proved to be terminating as follows: At the end of step 2 the value of is less than n, so if r = o the value of n decreases the next time step 2 is encountered. A decreasing sequence of positive integers must eventually terminate after a finite number of steps. Hence step 2 is executed only a finite number of times for any given n.

Our algorithm would succeed in proving that the inner loop terminates but would come out with no definite answer for the outer loop. Because in this particular problem r is a remainder, it need not, in general, decrease monotonically. In this example we notice that the value of n is decreasing in the outer loop. Our algorithm can thus obviously be extended to solve such cases.

Consider a loop with a looppredicate of the form $w_1\ pw_2$, where w_1 and w_2 are variables which are bound in the loop. Basically, we compare in our algorithm the rate of changes of w_1 and w_2. It appea rs that if we could transform this loop into two loops such that in each loop only it may one of the predicate variables is bound & the other is free, then we may require lesser amount of heuristics for checking whether the slack function satisfies the condition 1. The transformation we look out for need only be termination preserving. In this connection an interesting transformation was found for a particular type of the inner loop, but its generalisation to for all types of loops with any assignments has been

evasive so far.



It appears that the generalised transformation must be on similar lines.

We have mainly concentrated on proving termination of inner most loops only. The question of termination of concentric loops has to be examined. Some attempt in this direction is given in Appendix 2.

In this thesis we have concerned ourselves only with the problem of proving that a given program terminates, but another useful question to answer would be "How soon would the algorithm terminate?" I should be useful to be able to give time estimation for the execution of such programs. Meyer & Richie (31) have found a pretty close bound on the time required to execute a program containing only "DO" type loops. This time estimation along with storage require--ments will be useful in comparing programs (49).

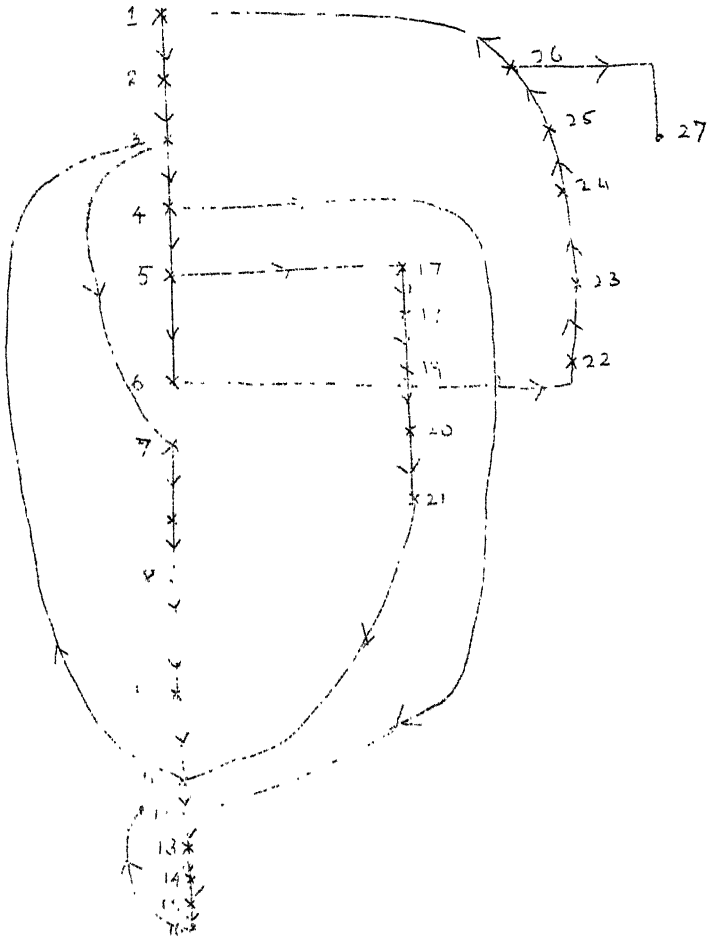
We now make an interesting conjucture, the proof for which has been evasive. That is "It is recursively undecidable to decide whether any given loopset is persistent or not."

Finally we have considered the termination of program for all values of input variables but it would be very useful to study how the different valuers of data affect the termination of a program. That is, we would like to find out the sets of values of data for which the program terminates.

- o O o -

A P P E N D I X - 1

Example 1: Consider the PS with standard interpretations.



1. L = 1
2. T = TN
3. IF (L=1) T_7
4. IF (L=2) T_{12}
5. IF (L=3) T_{17}
6. T_{22}
7. Q = H*F*CK
8. T = TN+H/2
9. X = XN+ Q/2
10. L = 2
11. T_3
12. Q = H*F*CK
13. T = TN+H/2
14. X = XN+Q/2
15. L = 3
16. T_{11}
17. Q =H*F*CK
18. T =TN-H
19. X =XN+Q
20. L = 4
21. T_{11}
22. Q = H*F*CK
23. TN = TN+H
24. IF(TN$\geqslant$11) I_{27}
25. XN = XN+ F
26. T_1
27. STOP

Note that the loopset with the first vertex at 1 and the last vertex at 11 is not persistent but it is observed that once the loopset is entered each loop in it is executed exactly once. The outer loop has an EXP at 24. The predicate variable TN is affected in each loop of the inner loopset. Since we know that the loop in the inner loopset is exactly once, it is seen that TN remains unaffected even after the execution of the inner loopset but in the outer loop TN is found to be increasing.

It is easily seen that

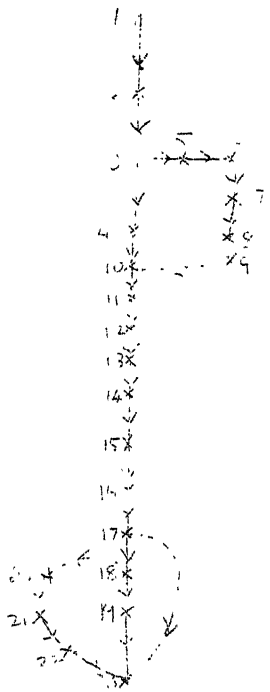
$f_n (w_1, w_2) = 11 - (TN_o + nH_o)$

$f_{n+1} (w_1, w_2) = 11 \ (TN_o = (n+1) H_o)$

$\therefore \forall n, \ f_n > f_{n+1}$

Hence, the program terminates.

Example 2:



1. ZNI = NI
2. H = (XUP - XLO)/ZNI
3. IF (INDEX=1) $\mathbb{T}_5$
4. $\mathbb{T}_{10}$
5. S = o
6. ZN = N
7. X = XLO - (ZN- 2.)*H
8. S = S + A
9. INDEX = 2
10. S = S+T
11. T=0
12. ZN = N
13. X = XLO + ZN*H
14. T = T + B
15. AREA = (C+D+2S+4T)*H/3
16. DIFF = ABS(1. - DAREA/SAREA)
17. IF (DIFF < ERR) $\mathbb{T}_{23}$
18. IF (PDIFF < DIFF) $\mathbb{T}_{20}$
19. $\mathbb{T}_{23}$
20. NI = 2*NI
21. PDIFF = DIFF
22. $\mathbb{T}_1$
23. STOP

Note that the loopset is not persistent but it is observed that the L^1 is executed, if at all, only once and hence L^2 is the loop for the termination of which we have to test. The EXF's are at 17 & 19.

Consider the EXP at 17

w_1 = DIFF; w_2 = ERR

It is easily seen that

$$f_n(w_1,w_2)=AB\ S(1-DAREA/((C+D+2S_o+2nH_o+2B_o(n+2)*(XVP_o- XLO_o)*6*2^nNT_o - ERR_o$$

$$f_{n+1}(w_1,w_2) = ABS\ (1-DAREA/((\ C+D+2S_o + 2nH_o+ 2H_o + 2nB_o + 6B_o)\ (XUP_o- XLO_o)\ 6*\ 2^{n+-}*NI_o - ERR_o$$

Now, if we assume f_n and f_{n+1} take only integral values then

$$\forall n,\ f_n > f_{n+1} \text{ if } B_o+H_o > NI_o.$$

The EXP at 17 can also be seen to result in the same expression. Note that in this program both variables in the loop predicate are bound in the loop.

....

A P P E N D I X 2

Termination of concentric loops

Let us assume a loop $L^{(2)}$ encloses an innermost loop $L^{(1)}$ $(=x_{i1}, x_{i2}, \cdots x_{ik}, x_{i1}; \mu_1 = x_{i1}, \cdots x_{ik})$ only. (see figure below)

If we do not allow any intersecting loop sets in our PS then it is clear that there can be only one entry point (x_{i1}) and one exit point (x_{ik}) from and to the outer loop, to the inner loop. Let us assume the innermost loop to be terminating. Let the set of variables bound in $L^{(2)}$ be $(v_{j1}, v_{j2} \cdots \cdots v_{jn})$ and the set of bound variables in $L^{(1)}$ be $(v_{j1}, u_{j2} \cdots u_{jm})$. Let $V_2(x_i)$ be the set of bound variables in $L^{(2)}$ which affect the loop predicate of $L^{(2)}$ at the vertex x_i. Let V_1 be the set of bound variables m $L^{(1)}$ which affect the variables in the loop predicate at x_i. Now if $V_1 \cap V_2 = \emptyset$ then it is clear that the innerloop has no effect on the termination of the outerloop. Now if $V_3 = V_1 \cap V_2 \neq \emptyset$ then by replacing the loop L_1 by a series of expressions corresponding to these variables in $R(u_1^n)$ we can simulate the effect of execution of the inner loop

n times. Note that the variables in these expressions take values just prior to the entry point of the innerloop and give the same value as the execution of the innerloop n times would do. Now.if the termination of the outer loop, with the above expressions replacing the inner loop, (so that the loop $L^{(2)}$ is transformed into an innermost loop) can be proved to be terminating irrespective of the index for the number of times the innerloop is executed then the outer loop terminates. We notice that every time the outer loop is entered, the number of times the inner loop is executed may be different. This is obviously a much stronger condition than necessary. This condition may be useful if some additional information regarding the number of times the inner loop is executed is given. The extension to the case when the loop enclosed is a member of a loop set, assuming the inner loop set is persistent and terminating is immediate.

An example to clarify the technique is given below:-

Example 1:- Consider the PS given below with standard

1 N=0 interpretation for symbols.

2 T = 0

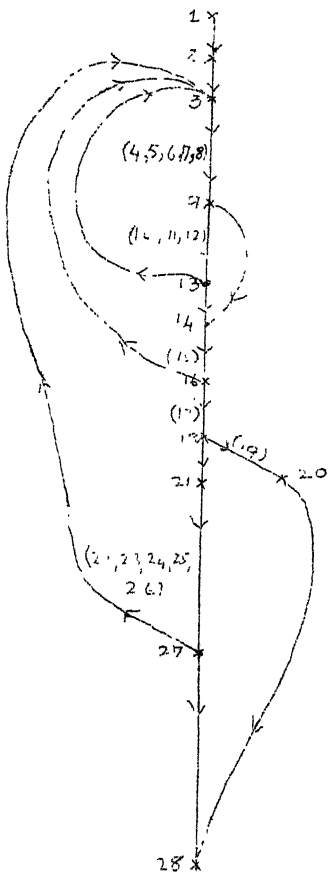
3 T1 = T+DT/2

4 RPMI = RPM+Q1/2

5 T2 = T+DT/2

6 T3 = T+DT

```
 7 RPM3 = RPM + Q3
 8 RPM = RPM + A
 9 IF(P=1.) ↑14
10 T = T+DT
11 SPEED = RPM
12 J = J+1
13 IF(T+DT/2 < .01) ↑3
14 T 0 0.0
15 N = N+1
16 IF (N≠75) ↑3
17 P = P+1
18 IF (P ≤175) ↑21
19 N = N-1
20 ↑28
21 N = 0
22 AVN = 0
23 AVI = AVI+K*AMP
24 AVN = AVN+K*SPEED
25 ACTQ=-C*AVI-A+B*AVN-TORQ+ACTQ
26 PRINT T
27 IF (ACTQ ≥ DTQ) ↑3
28 STOP
```



The flow-graph for the PS is given above. Note that the graph contains interesting loopsets but it is observed that

every loop set is persistent.

Let $L^{(1)}$ = 3-8, 9, 13, 3

The EXP's are at vertex 9 & 13

Consider the EXP at 9; $w_1 = P$; $w_2 = 1$

$\therefore f_n (w_1, w_2) = P-1$;

$f_{n+1} (w_1, w_2) = P-1$;

Hence, this function does not satisfy condition 1. We can not conclude any thing at this stage.

Consider the EXP at vertex 13.

$w_1 = T+DT/2$; $w_2 = 0.01$

$f_n (w_1, w_2) = .01 - (T+DT/2*(n+1))$

$f_{n+1} (w_1, w_2) = .01 - (T+DT/2 *(n+2))$

$\therefore f_n - f_{n+1} = DT$ o

$\therefore L^{(1)}$ terminates thro' 13.

Consider $L^{(2)}$ = 3-9, 14-16,3

EXP is at 16;

p is $\neq$ on (16, 17) = 0;

$w_1 = N$; $w_2 = 75$

$\therefore M = N - 75$ or $75-N$

$f_n (w_1, w_2) = N + n - 75$ or $75 - (N+n)$

$f_{n+1} (w_1, w_2) = N+ (n+1) - 75$ or $75 - (N+(n+1))$

$\therefore f_n(w_1, w_2) - f_{n+1} (w_1, w_2) = 75-(N+n)$

$-75 +(N+(n+1))$

>0

$\therefore \forall n,\ f(w_1, w_2) - f_{n+1}(w_1, w_2) > 0$ and $\leq f_n(w_1, w_2)$

$\therefore L^{(2)}$ terminates thro' 16

Consider $L^{(3)} = 3\text{-}16, 3$

This can also be shown to terminate thro' 16

Consider $L^{(4)} = 3\text{-}18, 21, 27, 3$

The EXP's are at 18 & 27

$L^{(4)}$ enclose $L^{(1)}$ & $L^{(2)}$

None of the assignments in $L^{(1)}$ or $L^{(2)}$ affect the loop predicate at 18. Hence, we can ignore the entire set of statements in $L^{(1)}$ and $L^{(2)}$. It is then easily seen that $L^{(4)}$ terminates thro' 18.

Consider the EXP at 27; w_1 = ACTQ; w_2 = DTQ

The assignment statements at vertex 8 & 11 affect the loop predicate 27. Both of these statements are contained in $L^{(1)}$ and $L^{(2)}$. Let r_1 and r_2 be the total number of times these two statements are executed before leaving the $L^{(2)}$ and entering $L^{(4)}$ at vertex No.21, when $L^{(4)}$ is executed n and n+1 times w.r.t vertex 27, respectively. Therefore, for n and n+1 the iterations of $L^{(4)}$ w.r.t vertex 27 the loop $L^{(1)}$ and $L^{(2)}$ can be replaced by the following equations.

SPEED = RPM + r_1*A and SPEED = RPM + r_2*A

RPM = RPM + r_1*A and RPM = RPM + r_2*A

$$\therefore f_n(w_1, w_2) = - C*n(AVI + n * k * AMP) - A* n$$
$$+ B * n * (AVN + n * k * (RPM + r_1 A))$$
$$- n * TORQ + ACTQ$$

$$f_{n+1}(w_1, w_2) = -C * (n+1) (AVI +(n+1)* k * AMP) - A* (n+1)$$
$$+B *(n+1)*(AVN+ (n+1)* k *(RPM + r_2*A))$$
$$- (n+1)* TORQ + ACTQ$$

(In both the above equations the values assigned to variables are those before just entering the entire set of loop In general, we will not be able to decide if $\forall n$, $f_n > f_{n+1}$, It is easily seen that $r_1 = r_2$. In this particular example

$$\therefore f_n - f_{n+1} = -C*n*AVI - k*n*2C*AMP - A*n +B*n*AVN$$
$$+B*n^2*k*RPM + B*n^2*k*r_1*A - n*TORQ + ACTQ$$
$$+C*(AVI)*(n+1) + C*(n+1)^2*kAMP + A*(n+1)$$
$$-B*(n+D*AVN - B*(n+1)^2 *k*RPM - B*(n+1)^2*k*r_1*A$$
$$+(n+1)*TORQ + ACTR$$
$$= C*AVI + 2*k*C*n*AMP + C*k*AMP + A-B*AVN$$
$$- 2*B*n*k *RPM - B*k*RPM - 2*B*n*k*r*A$$
$$-B*k*r_1*A + TORQ$$

Again, this expression is not independent of r_1 and hence we can not say anything about its termination thro' the EXP at vertex 27.

...

B I B L I O G R A P H Y

1. J.McCarthy - "A basis for mathematical theory of computation" - WJCC May 1961.

2. "Computer programs for checking mathematical proofs" - Proc.of American Mathematical Society - April 1961.

3. "Towards a mathematical science of computation" Proc. ICIP--1962

4. "Correctness of computer for arithmetic expressions" Stanford University, A.I. memo No.40 April 14, 1966.

5. "Problems in Theory of Computation" Stanford University, AI Memo 28.

6. "Recursive functions of symbolic expressions and their computation" Comm. ACM, April 1960.

7. D.C.Cooper - "Mathematical proofs about computer programs" Machine intelligence - V1. Ed.D.Michie.

8. "Some transformations and standard forms of graphs, with applications to computer programs". Machine intelligence.V-2

9. "Equivalence of certain computations" Computer Journal p.45 1966.

10. Letter to the Editor on Bohm & Jacopini's paper Vol.10, Aug.1967 IEEE Trans on EC

11. Y.I.Ianov - "The logical schemes of Algorithims" Problems of Cybernetics-V1-Pergamon Press

12. S.Igarashi. "An axiomatic approach to the Equivalence of Algorithims with Applications." Report of the computer centre University of Tokyo p 1-103. vol.1.No.1--April-Sept 1968.

13. "On the equivalence of programs represented by Algol like statements". Report of the computer centre University of Tokyo, Vol.1, No.1 p.103-118,April-Sept. 1968.

14. C.A.R.Hoare "An axiomatic basis for computer programming. Comm.ACM. Vol.12, No.10, p.576-580, Oct.1969

15. A.P.Ershov "Operator Algorithims1" - Problems of Cybernatics Vol.III.Pergamon P ress

16. M.S.Paterson. "Equivalence Problems in a model of Computation" Ph.d. thesis, University of Cambridge, Trinity College -Aug.1967.

17. "Program Schemata" p.18, Machine Intelligence V.III.

18. E.Engler "Algorithmic properties of Structures" Mathematical systems theory Vol.1, No.3 p.183-195 1967.

19. "Remarks on the theory of Geometrical construction" Lecture notes in Mathematics-- The Syntax & Semantics of infinitary languages--Ed.J.Barwise - 1968.

20. "Formal Languages.: Automata & Structures" Markham Publishing Company 1968.

21. D.M.Kaplan "Regular Expressions and Equivalence of programs", Journal of Computer & System Sciences 3, p.361-386, 1969.

22. R.W.Floyd "Assigning meaning to programs" "Proc.Symp. Appl.Math.Amer Nath Soc.Vol.1, 1967,p.19-32

23. Z.Manna "Properties of programs & First order predicate Calculus" JACM, Vol.16, No.2 April 1969 p.244-255

24. "The correctness of programs" Journal of Computer & System Sciences Vol.3 No.2 May 1969.

25. "Formalisation of properties of programs "Stan-ford University, Memo No.AI-64,June 1968.

26. Amir Pnueli "Formalisation of properties of Recursively defined functions" Stanford University, Memo No.AI 82, March 12, 1969.

27. Z.Manna "The validity problem of the 91-function Stanford University, Memo No.AI-68, Aug.19,1968

28. S.K.Basu "Transformations on directed graphs" Journal of Combinatorial Theory 8, 1970.

29. C.Berge. "Theory of graphs and its applications" Wiley, New York, 1962.

30. D.E.Knuth. "Art of Computer programming, Fundamental Algorithims V.1, Add.Wesley, - 1968.

31. A.R.Meyer & D.M.Richie. "Computational Complexity and program structure" 22nd National proceedings of the CACM - 1967.

32. R.T.Prosser. "Applications of Boolean matrices to the Analysis of programs" Proc.EJCC No.16, 1959, p.133

33. R.B.Marimont "A new method of checking consistency of precedence matrices" p.164, JACM 1959

34. S.Warshall "Theorem on Boolean Matrices" JACM 1962.

35. J.N.Nievergelt. "On the automatic simplification of computer programs", p.366, vol.8, No.6, CACM June 1965.

36. C.J.Maloney "A method for syntactic error analysis of computer programs" CACM, 1963, p.58

37. M.E.Senko "A control system for logical block diagnosis with data loading" CACM 1960, p.236

38. E.A.Voorhees "Algebraic formulation of flow diagrams

39. H.E.Ferguson & E.Berner "Debugging systems at the source language level" CACM 1963,p.480

40. L.Krider "A flow analysis algorithim" JACM,11,1964 p. 429-438.

41 Bohm & Jacopini "How diagrams, Turing machines, & languages with only two fomation rules" CACM 1966 p.366.

42. R.M.Karp "A note on the application of graph theory to digital computer programming" Information & Control 3, 179-190,1960.

43. H.Maekawa "Automatic flow charting" Information processing in Japan Vol.8, 1968

44. M.L.Minsky "Computation: Finite and Infinite Machine" Book, Prentice Hall, 1967.

45. M.Davis "Computability and Unsolvability" Mc Graw Hill 1958.

46. Hopcroft & Ullman "Formal languages and their relation to Automata", Addison Wesley, 1969

47. Mendelson E."Introduction to mathematical logic" Van Nostrand - 1964.

48. Church A. "Introduction to mathematical Logic" Vol.1, Princeton University press - 1956.

49. Hartmanis & Stearns. "On the computational complexity of algorithims" Trans Amer Math Soc 117, p.285-306.